Future Generation Computer Systems 101 (2019) 262–271

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Meteor: Optimizing spark-on-yarn for short applications

Hong Zhang [a,*], Hai Huang [b], Liqiang Wang [c,*]

[a] School of Cyber Security and Computer, Hebei University, Baoding, Heibei 071000, China
[b] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
[c] Department of Computer Science, University of Central Florida, Orlando, FL 32816, USA

HIGHLIGHTS

- Design a new scheduler which takes data locality and resource usage into account when allocating containers.
- Design a one-thread mode for spark, where all tasks of a short application are computed on just one thread.
- Design a one-container mode which deploys one container but with multiple virtual cores.
- Design a submitter framework with a reasonable number of AM containers to quickly bootstrap short applications.
- Design a profiler for Spark which uses bytecode instrumentation.

ARTICLE INFO

Article history:
Received 14 May 2018
Received in revised form 13 May 2019
Accepted 31 May 2019
Available online 19 June 2019

Keywords:
Spark
Short Application
Scheduling
Time-critical
Resource-sensitive

ABSTRACT

Due to its speed and ease of use, Spark has become a popular tool amongst data scientists to analyze data in various sizes. Counter-intuitively, data processing workloads in industrial companies such as Google, Facebook, and Yahoo are dominated by short-running applications, which is due to the majority of applications being mostly consisted of simple SQL-like queries (Dean, 2004, Zaharia et al, 2008). Unfortunately, the current version of Spark is not optimized for such kinds of workloads. In this paper, we propose a novel framework, called Meteor, which can dramatically improve the performance for short-running applications. We extend Spark with three additional operating modes: one-thread, one-container, and distributed. The one-thread mode executes all tasks on just one thread; the one-container mode runs these tasks in one container by multi-threading; the distributed mode allocates all tasks over the whole cluster. A new framework for submitting applications is also designed, which utilizes a fine-grained Spark performance model to decide which of the three modes is the most efficient to invoke upon a new application submission. From our extensive experiments on Amazon EC2, one-thread mode is the optimal choice when the input size is small, otherwise the distributed mode is better. Overall, Meteor is up to 2 times faster than the original Spark for short applications.

1. Introduction

Apache Spark [1,2] is an open source distributed computing framework to handle big data problems. Due to its high efficiency, versatility, and ease of use, Spark quickly builds a large community around it, and there is an increasing trend that Spark will replace other big data platforms such as Hadoop [3] and Storm [4].

Although Spark has its own standalone cluster manager, which provides almost all the resource management features, Hadoop Yarn is more mature, reliable, and secure for real-world deployments and can be used in conjunction with Spark. The common resource management of Yarn allows one to centrally configure and dynamically share the same pool of resources with other computing platforms like Hadoop MapReduce. Furthermore, one can choose a number of executors on each node to execute instead of an executor on every node in the cluster for each application. More importantly, Yarn provides a pluggable scheduler framework, which already has two built-in schedulers: CapacityScheduler and FairScheduler.

The major feature of Spark is its in-memory computing platform that speeds up the process of data crunching, especially for iterative applications. Spark allows applications to run up to 100 times faster by caching data in memory and 10 times faster even when disk is accessed compared with Hadoop MapReduce [1]. Although Spark is fast enough to solve big data problems, in reality, a majority of workloads are completed within a *short* amount of time [5–7]. There are two reasons: (1) we are getting much better at building gigantic clusters (especially in large companies), so even a large amount of input data can be dwarfed when massive

* Corresponding authors.
E-mail addresses: hzhang@hbu.edu.cn (H. Zhang), haih@us.ibm.com (H. Huang), lwang@cs.ucf.edu (L. Wang).

https://doi.org/10.1016/j.future.2019.05.077
0167-739X/

amount of hardware can be utilized efficiently in parallel; (2) many workloads are simple SQL-like queries for structured data such as Spark SQL and Hive. To handle the latter case, Hadoop MapReduce specifically added a new processing mode called Uber mode that runs all Map and Reduce tasks in a single container. Unfortunately, this mode is still not very efficient [8], and more importantly, there is no corresponding mode in Spark. The major disadvantages of the current Spark on Yarn framework are listed as follows:

- Hadoop Yarn does not consider data locality for short-running applications, which incurs additional overheads when shuffling data between Data Nodes. This drastically affects the performance for such applications.
- The request, deploy, setup, launch, and tear down overheads of the Application Master are much more prominent in short-running applications.
- Piggybacked communications between the Application Master and the Resource Manager are extremely inefficient when asynchronously waiting for resource responses.
- No support of Uber-like mode in Spark results in not only inefficiency but also resource waste for short-running applications.
- Moreover, there is no clear decision-making mechanism to decide how to best run a short application, which leads to bad configurations or misuse of Uber-like modes.

Short application (or short job) has been studied in the past [5–7]. However, the definition of short application is still unclear to guide their implementation and execution. In our previous work [8], we took resource usage and cluster size into account for our definition of a short application and designed a submitter system to help user decide which application is a short application and how to execute it efficiently on Hadoop. In this study, we show an efficient short application optimization framework for Spark, called *Meteor*, which can improve the performance and help user run a short application in one of three modes we add: one-thread, one-container, and distributed. This paper is a significant extension to our previous study [8] for Hadoop. Our contributions of this paper are summarized as follows:

- We design a new scheduler that takes data locality and resource usage of each node into account when allocating containers. Instead of waiting for node updates from Node Manager (NM) when an Application Master (AM) asks for containers from Yarn, our scheduler responds immediately by considering the resource status of each node and data location information. Our algorithm not only reduces the communication cost between the AM and the Resource Manager (RM), but also reduces resource contention and data transferring.
- We design one-thread mode for spark, where all tasks of a short application are computed on just one thread. It shows better performance under certain scenarios especially when cluster is busy.
- Another mode we design for short applications is one-container mode, which deploys one container but with multiple virtual cores. This mode takes full advantage of context sharing within a single container and running multiple tasks in parallel. It also shows better performance in our experiments under certain scenarios.
- Due to the expensive process of setup, deploy, and launch an AM from scratch for every short application, we allow one to pre-allocate an AM pool with a reasonable number of AM containers to quickly bootstrap short applications.

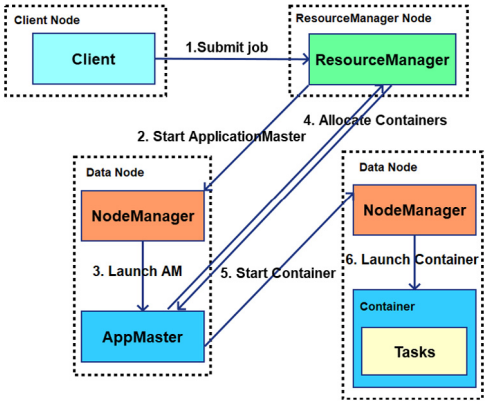


Fig. 1. Hadoop Yarn application submission [9].

- In order to decide which mode is the most efficient for newly submitted short applications, we design a new submitter framework, which has a decision maker that monitors the resource usage of the cluster and predicts the performance of the three modes.
- We also design a profiler for Spark that uses bytecode instrumentation technique to profile applications for useful information with lower overhead.

The rest of this paper is organized as follows. In Section 2, we give an introduction to the application submission process in Hadoop Yarn. Then we discuss our design and implementation of Meteor in Section 3, which supports three new modes: one-thread, one-container, and distributed. In Section 4, we show experimental results. Section 5 gives a review of related work. Finally, Section 6 concludes remarks.

2. Background

To discuss the disadvantages of the original RM in Yarn, we first introduce some background knowledge of the application submission process in Yarn, where submitting an application will trigger the launching of an AM so that it can request containers for its tasks' execution. There are 6 steps in this process, as shown in Fig. 1.

1. *Application submission:* A client first contacts the RM to obtain a new application id. After checking the specification of the application, the client copies input splits, the application's jar file and other resources to HDFS. Then the application is submitted to the RM.
2. *AM allocation:* When the RM receives the request, it allocates a container to deploy an AM for this application.
3. *Launching AM:* The corresponding NM launches a container to run the AM for this application. After that, the AM pulls all files from HDFS, which includes input splits, the application's jar file, and configurations.
4. *Request containers:* Then the AM requests containers from the RM by piggybacking with a heartbeat message. When the AM receives the resources from the RM, it schedules tasks based on data locality.
5. *Task assignment:* The AM communicates with NMs to launch and monitor containers to run tasks.
6. *Task execution:* Each task is scheduled by a Task-Scheduler (TS) to a container to execute.

From the steps we mentioned above, data locality is only considered when the resource is assigned by the RM, which is too

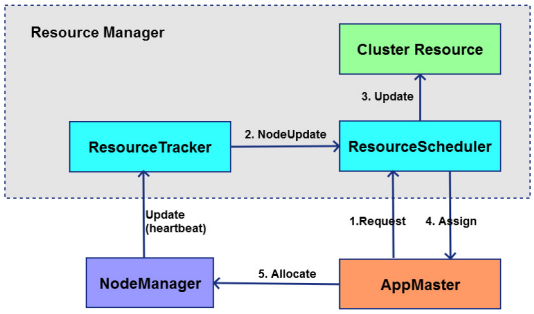


Fig. 2. Resource request in Hadoop Yarn [9].

late, since the designated node may not have the input data for this short application. Another problem is that the scheduler does not consider the resource utilization ratio of each node, so that it often bin packs all containers to a few nodes. This may cause severe resource contention for short applications. Furthermore, there does not exist a Uber-like mode for Spark to execute all tasks on a single node. Last but not least, the process to request, allocate, and launch an AM is time-consuming for short applications.

3. Design and implementation

To decide how to execute a short application more efficiently on Spark is very difficult: running all tasks in a single container or executing them distributively in the cluster. Executing all tasks of all stages in one container not only reduces a large amount of time to request, deploy, and launch executors from different computing nodes, but also avoids burdensome remote shuffling and communications with one another; however, due to parallelism, the performance of using one container is sometime slower than distributing all tasks in the whole cluster, which can fully use the resource in the cluster, if there are a bunch of small tasks to be executed and the file size is relatively large.

Therefore, we design a system that includes three modes: one-thread, one-container, and distributed. One-thread mode executes all tasks in one container with one thread, which utilizes resource as little as possible. One-container mode still uses one container to run all tasks but with multiple virtual cores, which increases the parallelism degree in a multi-threading way. The distributed mode is to distribute all tasks in the cluster uniformly to take full advantage of the cluster resource so that it reduces execution waves of the whole application as far as possible.

From our experiments, we found that to setup and launch AM is very expensive for a short application no matter what kind of mode to implement. Thus, we design a submitter framework by Spring Hadoop [10] that reserves an AM pool used for short applications in order to reuse AM container rather than to request it each time for a short application.

3.1. Distributed mode

To reduce memory, CPU, and disk I/O contentions in each computing node and move computation close to data as much as we can, we redesign the resource scheduling strategy in Yarn, which is more considerate to data locality and resource utilization so that we can execute each task near to its data in a relatively idle machine. This scheduler also speedups data transmission between Data Nodes in the shuffling processes as a result of uniform task distribution.

Fig. 2 demonstrates the workflow to request containers from the original RM of Yarn.

1. *Container request:* In order to ask containers to execute tasks, the AM sends a request to the RM. This request is sent periodically as a piggyback of heartbeat, which contains information of new request, a list of released containers, and a set of blacklist nodes.
2. *Node update:* After the RM receives the heartbeat, it puts this request of container into a corresponding queue and waits for the NM to update its resource status by a NODE_STATUS_UPDATE event.
3. *Container assignment:* The Resource Scheduler (RS) updates the resource information and allocates available resources on the Worker Node if it has available resource to use. The RS will select the container request sequentially in the front of the request queue to assign if the available resource on this Worker Node can satisfy the container request.
4. *Container allocation:* Finally, the AM requests the NM to launch containers to run tasks of each stage and the NM will also register these containers to the RM for updating.

From the above discussion, we notice that resource cannot be allocated to the AM until the RM receives the NODE_STATUS_UPDATE signal from one NM, even there exists ample idle resource in Worker Nodes. Another big problem is the frequent communications between the AM and the RM, which causes that the AM has to wait at least two heartbeats to receive resource. Last but not least, the skew task scheduling is liable to cause resource contention and data transferring due to bad data locality for the first-stage tasks. Although the scheduling algorithm introduced above is not a big problem if the input data are large, it is time-consuming and a severe problem for short application if ignoring data locality and resource utilization of each Worker Node, especially when there is enough resource to schedule in the cluster.

Algorithm 1 Scheduler algorithm for distributed mode

Input: *request*, *nodes*
Output: *response*

```
types = {NodeLocal, RackLocal, ANY}
/* Priority order: NodeLocal > RackLocal > ANY */
for each type in types do
  for each task in request do
    Decide which resource is the current dominant resource;
    Sort nodes by available dominant resource in descending order;
    for each node in nodes do
      container = getResource(task, node, type);
      updateClusterResource(container);
      if (container is not null) then
        response.add(container);
        request.del(task);
      end if
      if (request is empty) then
        return response
      end if
    end for
  end for
end for
return response
```

Therefore, we design a brand-new algorithm that considers the resource usage in each Worker Node in order to schedule tasks in a more uniform and efficient way and also assigns tasks to the relatively idle nodes at first, as shown in Fig. 3. Firstly, after the RM receives the request of containers from the AM, it sends a CONTAINER_ STATUS_UPDATE event to the RS to check whether there is enough resource for this request. Secondly,

rather than waiting for updating their resource status to the RM from Worker Nodes, we design an in-time response mechanism, which asks for resources directly from the Cluster Resource. The Cluster Resource is a structure maintained by the RM that stores all resource status for each Worker Node, and decides the scheduling strategy by Algorithm 1. After receiving the specific assignment of resources from the Cluster Resource, the RS sends back the response to the AM immediately without any waiting. The resource status is updated by heartbeat so that the Cluster Resource can maintain the latest resource status. At last, the AM asks the corresponding Worker Nodes to launch the containers and report them to the RM, which is the same as the original Yarn. The frequency of heartbeat is one second by default. In our design mentioned above, we only need one heartbeat to respond the resource request, which means we can save more than one second compared with the original Scheduler.

In Algorithm 1, we improve the original CapacityScheduler algorithm that sorts all Worker Nodes by available dominant resource from upper to lower so that the relatively idle nodes will be assigned first. Dominant resource is the resource with the maximum utilization rate in the cluster. Here we only consider CPU and memory that are easily to detect and analyze, and have distinct influence on the performance. Note that we use virtual core to measure the CPU usage in Amazon EC2 and our definition of dominant resource is different from the definition of dominant resource in [11] for each user. The original CapacityScheduler schedules tasks by a Bin-Packing algorithm, which uses as few as possible Worker Nodes to allocate all containers. This does not take data locality into account for short applications, and may cause high possibility of resource contention. Our algorithm employs a round-robin method that has a global view of the whole cluster.

There are three types of container requests (*i.e.*, NodeLocal, RackLocal, and ANY) according to the relationship between request location and resource location. Request location is the recommended location by Spark, which is always the data location. The resource location is the available resource location selected by the RM. *NodeLocal* is the type that the request location is the same as the resource location; *RackLocal* means the preferred node and the resource node are in the same rack; and *ANY* is that the RM can designate any resource node. In Algorithm 1, the order of priority is NodeLocal $>$ RackLocal $>$ ANY. For each container, we assign a NodeLocal node firstly. If the NodeLocal resource is not enough, we assign a RackLocal instead, otherwise an ANY node is selected until this request is satisfied. After each allocation, we recalculate the current dominant resource after each assignment to ensure that the idlest nodes can serve the next request. From Algorithm 1, we consider data locality in preference to resource utilization due to the importance of data locality and the reasonable data distribution.

As we discussed above, our distributed mode not only takes the data locality into account, but also reduces the chance of resource contention and computation skew problem. Moreover, our algorithm also avoids bandwidth congestion and improves the communication efficiency between the AM and the RM.

3.2. Spark one-thread mode

As we discussed before, there exists an Uber mode in Hadoop that executes all Map and Reduce tasks in one container to reduce the overhead of requesting, launching, and deploying remote containers and avoid the expensive data shuffling between each other. However, Spark does not support the Uber mode. Although we can run Spark in a local mode, which uses non-distributed single-JVM driver for execution on the driver's node, this is just for testing, debugging or demonstration rather than real industrial deployment. Hence, we design a one-thread mode that runs

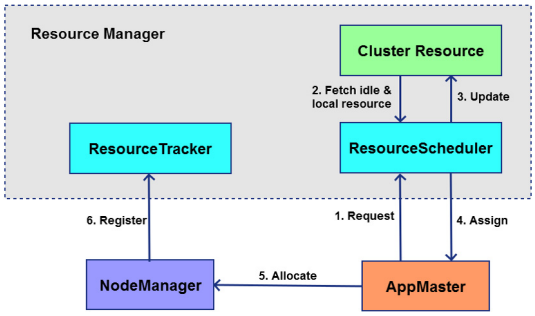


Fig. 3. Resource request in the distributed mode of Meteor.

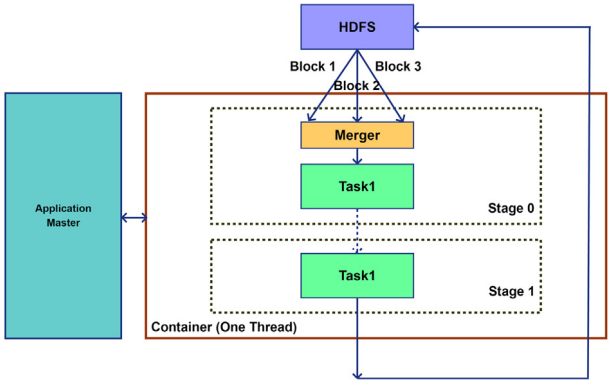


Fig. 4. Spark's one-thread mode.

all tasks in every stage just in one thread, which dramatically reduces the resource usage for a short application.

Fig. 4 demonstrates the workflow to execute a Spark application in the one-thread mode. The major modification is that we add the input data merger before task execution, which merges all input data into one partition, so that only one task is executed for each stage. This merger employs an inputstream to read all input data from HDFS, which does not cause any overhead. One-thread execution for the whole application minimizes the total resource usage, which is very suitable for a busy cluster and an application with small input data size. But the duration will increase with the increment of the input data size, and the cluster's resource is not fully utilized if it is relatively idle.

3.3. Spark one-container mode

Due to these inefficiency issues of the one-thread mode, we design the one-container mode that inherits the single container feature from the one-thread mode, but use multiple threads to execute tasks in parallel by multi-threading. The one-thread mode takes full advantage of the resource in one node to run all tasks of each stage concurrently using multi-threading technique. Fig. 5 shows a detailed workflow of the one-container mode. Thread-level parallelism makes the execution always faster than one-thread mode based on our experiments, and the number of cores in one container can be determined by min(*num_vcores*, *num_tasks*).

The major difference between Hadoop Uber mode and our one-container mode in Spark is that Hadoop Uber mode reuses AM to execute all Map and Reduce tasks. Due to the complexity of Spark applications that usually have more than 2 stages, we require another container besides AM to run tasks, which shows good performance in despite of more resource utilization. Another difference is Hadoop Uber mode cannot execute tasks in parallel, whereas our one-container mode allows.

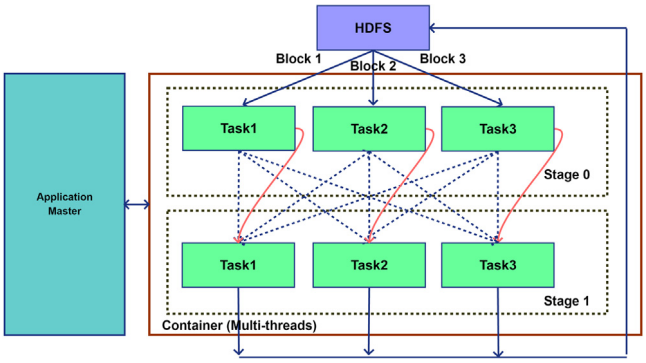


Fig. 5. One-container mode in Spark.

Fig. 6. The overhead of AM creation on WordCount by varying the number of files.

3.4. Resource-considerate application submitter

In our design, there are three different modes (distributed, one-thread, and one-container) to choose. Distributed mode scatters all tasks to multiple computing nodes to make all processes run in parallel; the one-container mode is to execute all tasks by multi-threading to avoid remote transmission and communication; one-thread mode only relies on a single thread to compute all tasks. According to our experiments, we found that when the input data size is small, it is better to use one-container mode, or even one-thread mode, otherwise it is preferred to use the distributed mode. However, how to precisely select the best mode to deploy a Spark short application is a grand challenge. This is a multi-objective optimization problem (MOOP) which not only needs to consider the time consumption but also the resource usage of each mode. The optimization problem is shown in Eq. (1). Θ expresses the three-mode decision space. α denotes the time-critical ratio, which is 0.8 by default due to the importance of execution efficiency. This ratio is configurable by users depending on the consideration of timeliness of each application. t_i^{app} is the execution time for mode i, which is evaluated by our performance model mentioned later, and $\max t_i^{app}$ is the maximum duration of three modes, r_i^{app} is the dominant resource utilization of mode i, and r^{avail} is the available dominant resource in the cluster. From Eq. (1), it is easy to find that the less the resource available in the cluster, the larger possibility to select one-container mode, even one-thread mode.

$$\underset{i\in\Theta}{\operatorname{argmin}}(\alpha \cdot \frac{t_i^{app}}{\max t_i^{app}} + (1-\alpha)\cdot \frac{r_i^{app}}{r^{avail}}) \tag{1}$$

Because the duration of a short application is really short, the overhead of setting up AM turns out to be very expensive, which is shown in Fig. 6. For WordCount, when the number of files is 4 and the file size is 10 MB, the overhead of AM launching is more than 40%. Therefore, we design a submitter framework that creates a reserved AM pool to reserve a reasonable number of containers specifically for short applications. We also design a performance model to decide which mode is the optimal one, considering the dominant resource usage in the cluster.

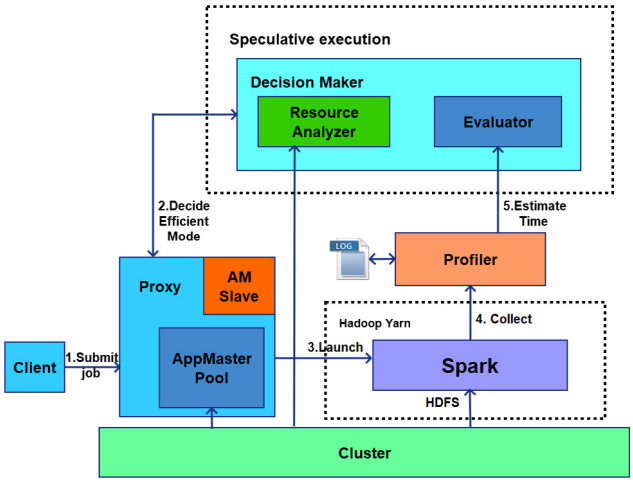


Fig. 7. The submission framework in Meteor.

Fig. 7 demonstrates the process to submit a Spark application. When the client submits an application, instead of requesting AM from the resource manager Yarn, we design a novel application submission framework using Spring Hadoop [10] to accept the application submission from the client. Our submitter contains three major components: (1) The client module is used to submit short applications to a proxy module and upload the related files such as configuration and jar files to HDFS. (2) The proxy module is responsible for accepting the application request from the client, and then allocates an AM from the AM pool that contains a reasonable number of AMs. (3) AMSlave module is the module to launch AM from the proxy module rather than Yarn, and run the "main" function for the Spark application. The workflow of application submission is shown in Fig. 7.

1. **Application submission:** The proxy module is launched when Hadoop Yarn starts, and creates an AM pool that contains a certain number of AMs in order to deploy short applications. The user can configure the number of AMs in the AM pool depending on the submission frequency of short applications. After the proxy starts up, the client can use the client module to submit a short application in our submitting framework.
2. **Decision making:** The proxy module is a daemon that waits for the requests from clients. The proxy consults the decision maker to decide which mode is the most efficient way according to the time estimated by the performance model and resource usage status from the resource analyzer.
3. **Launching AM:** If there exists history logs collected by our profiler for this application, the decision maker can make choice based on the duration and resource utilization of each mode in the history. Then the proxy launches one AM from the AM pool to execute it. Otherwise, the proxy will launch this application depending on the current resource usage in the cluster according to a conservative order *distributed* > *one-container* > *one-thread*.

Table 1
Notations used in the estimation algorithm.

t_{app}	The total duration for Application
t_{job_i}	The total duration for job i
t_{stage_i}	The total duration for stage i
t_{task_i}	The total duration for task i
t_{am}	The AM setup time
$t_{executors}$	All executors setup time
DOP	Degree of parallelism
$t_{schedule}$	The time to schedule task to executor
$t_{deserialize}$	Task deserialization time
t_{run}	The exact execution time of a task
$t_{serialize}$	Task serialization time
t_{result}	The time to fetch result data to driver
$t_{shuffle\_read}$	The shuffle read time
$t_{shuffle\_write}$	The shuffle write time
t_{cpu}	The real cpu computing time of a task
t_{idle}	The idle time of a task
d_i	Disk input rate
d_o	Disk output rate
b_i	Bandwidth
s_r	Average shuffle read of tasks
s_w	Average shuffle write of tasks

4. **Profiling:** Although Spark collects some execution metrics, this information is still insufficient to design an accurate performance model to evaluate the duration without running. In our framework, we design a Spark profiler, which employs a light-weight bytecode instrumentation tool, called ASM [12], to collect fine-grained information such as the duration of sub-phases and RDD operations within each task.
5. **Evaluation:** According to the logs collected by the Spark profiler, we use our performance model to estimate the duration of the application in three modes. The fine-grained performance model will be introduced in Section 3.5.
6. **Terminating:** When the application finishes, the proxy receives a notice from the AMSlave, it releases the AM, and returns it back to the AM pool.

3.5. Performance model

In order to select a more efficient mode to execute, we design a performance model, called Hedgehog, which is a white box evaluator to estimate the dataflow and cost of Spark applications based on the profiler logs, Spark logs, and Yarn logs.

$$\begin{aligned} t_{app} &= t_{am} + t_{executors} + \sum_{i=1}^{n} t_{job_i} \\ t_{job_i} &= \sum_{stage_j \in job_i} t_{stage_j} \\ t_{stage_j} &= \frac{\sum_{task_k \in stage_j} t_{task_k}}{DOP} \end{aligned} \tag{2}$$

$$t_{task} = t_{schedule} + t_{deserialize} + t_{run} + t_{serialize} + t_{result} \tag{3}$$

$$\begin{aligned} t_{run} &= t_{shuffle\_read} + t_{cpu} + t_{idle} + t_{shuffle\_write} \\ &= s_r/d_o + s_r/b_i + t_{cpu} + t_{idle} + s_w/d_i \end{aligned} \tag{4}$$

Table 1 gives the notations used in our performance model. Eq. (2) shows an estimation of the total duration of a Spark application. The AM setup time can be avoided due to our submitter framework. $t_{executors}$ is the time to deploy all executors in cluster's nodes, which is linear to the number of nodes to be used in each mode. For one-container and one-thread modes, there is only one container to be launched, which means that only one node needs to be deployed. For distributed mode, the number of used nodes relies on the number of tasks to be executed in the first stage if there is no repartition operation. The total execution time of an application consists of the duration of each job. Each job contains multiple stages, and each stage has a bunch of tasks to be executed. The degree of parallelism can be expressed by *DOP*, which indicates how many tasks executed in parallel. For one-container mode, *DOP* is the number of virtual cores in each executor. The one-thread mode only uses one virtual core during the whole application process, where *DOP* is 1. In the distributed mode, *DOP* counts on the total virtual cores of all containers.

Eq. (3) gives an evaluation of the duration for a task. The whole execution time of a task consists of 5 major sub-phases: *schedule*, *deserialize*, *run*, *serialize*, and *getting-result*. Since *schedule*, *deserialize*, *serialize*, and *getting-result* are related to the specific characteristics of the task setup and cleanup, the three modes have almost the same duration of these four sub-phases, which is proved by our experiments. Therefore, we only consider the *run* sub-phase to decide which mode should be the best to execute. The *run* sub-phase can be further divided into four sub-sub-phases: *shuffle-read*, *CPU-computing* (called cpu), *idle*, and *shuffle-write*, as shown in Eq. (4). *shuffle-read* is related to the shuffle read size, the bandwidth ratio, and disk input ratio that can be evaluated and detected by our Hedgehog system. *cpu* time is linear to the load of one task. Since the one-thread mode runs all tasks in a sequential way, which means that it need more time than other two modes for this sub-phase. *idle* time depends on the memory, CPU and disk I/O contentions during the task duration determined by the number of cores in each container.

4. Experiments

4.1. Experimental setup

We conduct all experiments on a cluster on Amazon's Elastic Compute Cloud (EC2) that consists of one Name Node and ten Data Nodes. The instance type we used is Amazon m5.2xlarge, which has 2.5 GHz Intel Xeon Platinum 8175 processors, 32 GB memory, 8 vCPUs, SSD-based instance storage for fast I/O performance, and up to 25 Gbps network bandwidth. Our experiment is based on Apache Spark 2.0, Apache Hadoop 2.3, and Java 7. To evaluate the performance of our new framework, we run four different benchmarks: WordCount, PageRank, K-Means, and SQL query. WordCount is a classic big data program that counts words from a large file, which was downloaded from Wikipedia database. PageRank is an algorithm used to rank websites. For PageRank, the input data are generated by "GraphGenerators" from Spark API, and the iteration number is set to 5. K-Means is a famous clustering method that clusters vector observations into k sets. For K-Means, we set the iteration number to 5, and the data are generated by Spark MLlib API "KMeansDataGenerator". SQL query is a "selection" operation that counts how many numbers are greater than 0 in one column, which also uses K-Means data. The performance of each benchmark is barely influenced by the distribution of input data.

4.2. Experimental results

4.2.1. WordCount

Fig. 8 shows the performance of Meteor and Spark with the number of files varying from 4 to 32 and the file size being set to 10 MB. We compare three new modes with the original Spark. When the number of files is 4, the one-container mode gains an improvement of 66.6% compared with Spark. This is because the one-container mode not only significantly avoids shuffling

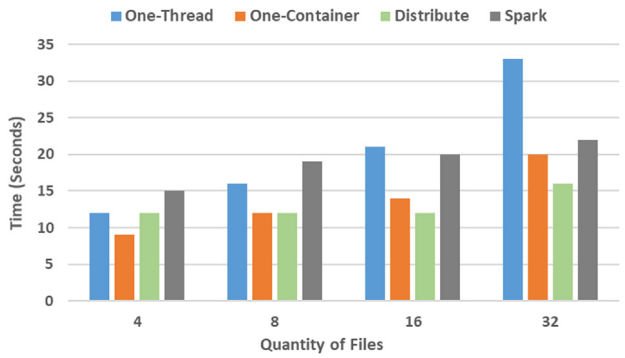


Fig. 8. The performance on WordCount when varying the number of files but fixing the file size to 10 MB.

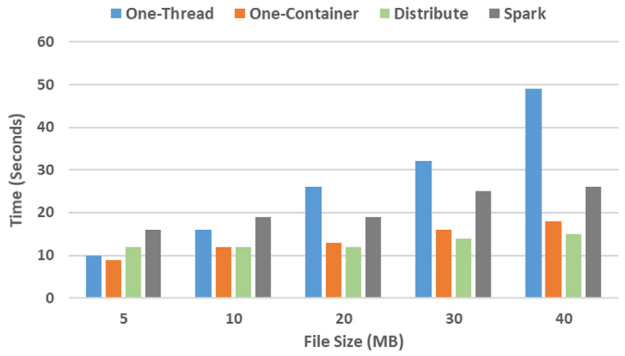


Fig. 9. The performance on WordCount by fixing the number of files to 8 but varying file size.

data between containers, but also reduces the containers' setup and deploy time compared with the distributed mode. However, when the number of files increases to 16, the distributed mode becomes a better choice since the higher level of parallelism can speed up the total performance and improve data locality, but with the cost of 4 containers. Fig. 8 demonstrates that as the number of input files grows, the distributed mode outperforms one-container mode even more. When the number of files is 32, the distributed mode is 25% faster than the one-container mode. Although the one-thread mode is more time-consuming than the other two modes, it is still a good choice when the cluster is short of resource and cannot allocate enough for the distributed and one-container modes. Note that even the one-thread mode is better than the original Spark when the number of files is small.

In Fig. 9, we vary file size from 5 MB to 40 MB but keep the number of files fixed at 8. As input files grow larger, the distributed mode also becomes increasingly better than the other two modes. This is because the large input data size increases the chance of CPU and disk I/O contentions compared with only one container. Another reason is that the one-container mode has to execute 2 waves (8 tasks/4 cores), which is more time-consuming than the distributed mode, which runs only one wave. When the file size is 80 MB, the distributed mode is 20% faster than the one-container mode, and 73.3% faster than the original Spark. As expected, the performance of the one-thread mode becomes worse when the file size increases.

Fig. 10 demonstrates the performance when the total input data size is fixed to 160 MB, but the number of files varies from 2 to 8. The distributed mode is the optimal choice regarding from performance's point of view, especially when the number of files is large. However, when the number of files is small, the one-container mode is also a good candidate, since it is only about 7%

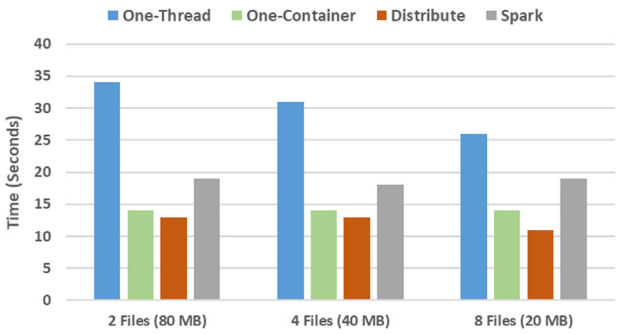


Fig. 10. The performance on WordCount when fixing the input size to 160 MB.

Table 2
Mode selection based on performance model.

Selection		Resource availability				
		20%	40%	60%	80%	100%
Num of files	2	1-Thread	1-Thread	1-Thread	1-Thread	1-Thread
	4	1-Container	1-Container	1-Container	1-Container	1-Container
	6	1-Container	1-Container	Distributed	Distributed	Distributed
	8	1-Container	Distributed	Distributed	Distributed	Distributed

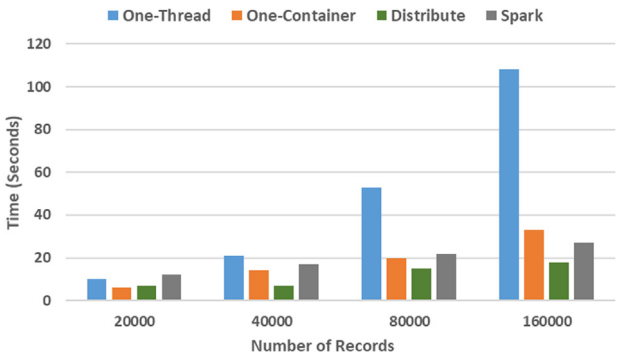


Fig. 11. PageRank performance with different numbers of records.

slower than the distributed mode, but only costs 1/4 of resource usage.

Table 2 shows the selection results when the resource availability varies from 20% to 100% and the number of files grows from 2 to 8. As the number of files is 2, one-thread mode is best no matter the available resource rate is from 20% to 100%. However, when the number of files is $\geq$ 4, the one-container mode is selected due to its multi-threading and less resource usage. Only when the quantity of files is 8 and the resource availability is large ($\geq$ 40%), The distributed mode outperforms the other two modes because of better parallelism. The overhead of our submitter framework is mainly caused by the decision-making process. The time consumption of this process depends on the size of the history logs, which are organized as json format. Due to our compact performance model shown in Section 3.4 and small log size for short job, the duration of decision-making is within 1 s, and the decision is sent back with resource allocation information, which does not generate any extra overhead costs.

4.2.2. PageRank

In Fig. 11, we execute another classic benchmark called PageRank. We vary the input data records from 20 000 to 160 000. Each block contains 10 000 records, therefore, the number of blocks changes from 2 to 16 when the number of records varies from 20 000 to 160 000. We found that its result is similar to that of WordCount, where the one-container mode is the optimal mode

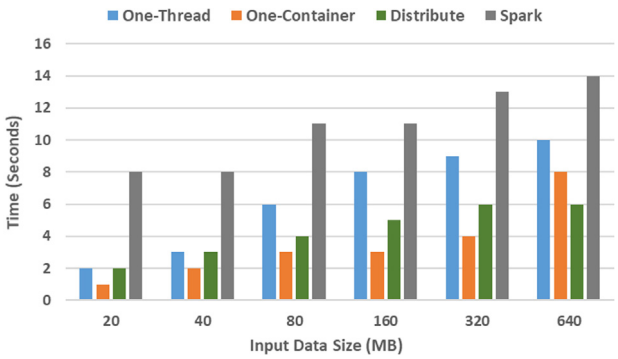


Fig. 12. K-means performance when varying input data size.

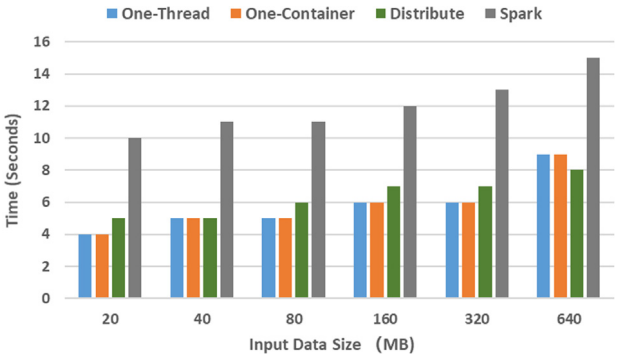


Fig. 13. SQL query performance when varying input data size.

when the input data size is small, and the distributed mode is better when the data size becomes large. We also found that the duration of one-thread mode increases drastically when the data size is bigger due to the more complex calculation and more stages in PageRank. Our one-container mode is twice as fast as the original Spark when the number of records is 20 000.

4.2.3. K-means

Fig. 12 shows the performance of K-means. The input data size is varied from 20 MB to 640 MB. An interesting thing is that the distributed mode outperforms the one-container mode only when the input data size is very large (more than 640 MB), because this benchmark does not require massive computation, and there is little CPU contention, which enhances the benefit gained by the one-container mode.

4.2.4. SQL query

Fig. 13 is another benchmark to execute a SQL query that counts numbers in the dataset to be greater than 0 with the K-means data. This benchmark is even lighter weight than PageRank, where the majority of the time consumed is in loading data from HDFS. We found that the one-thread mode is always a better choice even when the data size has grown to 640 MB. When the input data size is 20 MB, the one-thread mode is 1.5 times faster than the original Spark.

4.3. Cross-comparison for WordCount, PageRank, K-means, SQL query

From our experiments, it is worth noting that the lighter the application is, the better the performance of one-thread and one-container modes are. Due to the less resource contention for K-means and SQL query, one-thread or one-container is always a better choice even if the input data size is large, which is also better than the original Spark. However, if the application is resource-intensive (e.g., IO, CPU or Memory) such as WordCount and PageRank, the one-thread or one-container mode can gain the improvement only when the input data is very small. Therefore, we have to make a mode decision by performance model rather than experience.

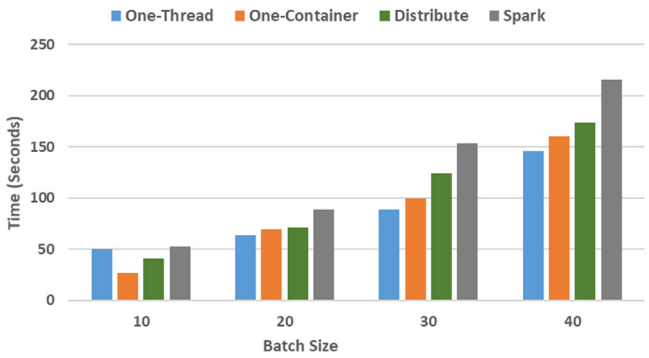


Fig. 14. WordCount performance when varying batch size.

4.4. Experimental results for batch applications

In this section, we discuss the performance of batch processing that executes a series of Spark applications in parallel. In reality, multiple tenants could share the same Spark cluster and submit applications simultaneously. It is crucial to detect what mode is the best mode when the cluster is busy. Here we use batch-submission to analyze the performance of short applications.

In Fig. 14, we vary the number of WordCount applications in a batch from 10 to 40. When the number of applications is 10, the one-container mode is the optimal mode because it not only requests few resources in the cluster, but also has multi-threading to speed up the work. However, when the number of applications is changed to more than 20, the one-thread mode becomes more efficient due to fewer resource requested and less contention in the cluster. The one-thread mode outperforms the distributed mode by 20% when the number of WordCount applications is 30. As the number of WordCount applications is 40, The one-thread mode can outperform the original Spark by 48%.

Fig. 15 demonstrates the performance of three modes for another benchmark, SQL query in batch-submission. The computation of SQL query is even simpler than WordCount so that even with one thread, Spark can compute all tasks very fast. From Fig. 15, even when the number of applications submitted is 10, the one-thread mode shows good performance, although it is still worse than the other two modes. However, when the number of applications is greater than or equal to 20, the one-thread mode becomes the most efficient mode and is 80% faster than the distributed mode at 40 applications. From experiments shown in this section, we found that when the cluster is relatively busy, the one-thread mode is always a good choice as it is more resource-efficient than the other two modes.

5. Related work

There are four research areas related to our system regarding optimizing performance of short applications for Spark: data-locality awareness [13–16], application scheduling [7,17–22], performance model [23–26], and multi-threading [27,28].

Data-locality awareness. Hammoud et al. [13] proposed a locality-aware Task Scheduler based on data locality and size, which has significant improvement when the data are skewed.

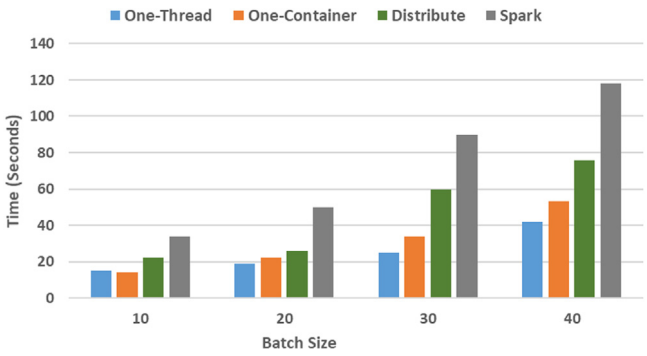


Fig. 15. SQL query performance when varying the batch size.

Zhang et al. [14] introduced a next-k-node scheduling (NKS) algorithm that reserves nodes to satisfy data locality. Maestro [15] designed a scheduling algorithm to schedule all tasks in two steps to achieve better data-locality. Marco et al. [16] presented a mixture-of-experts approach to model the memory behavior of Spark applications in order to determine how many tasks to co-locate on the same host to improve system utilization and throughput.

Application scheduling. Elmeleegy [7] introduced a system called Piranha to avoid saving intermediate results to disk and provide a fault-tolerance mechanism. Yao et al. [17] designed a scheduling algorithm considering the size of applications to decrease average application response time by adjusting resource sharing among users dynamically. Yan et al. [18] implemented an optimized Hadoop that employed a push-model assignment mechanism instead of the pull-model to reduce the initialization and termination time of a application. Chen et al. [19] proposed a pre-scheduling framework to predict stragglers to minimize the processing latency. Yan et al. [20] designed a new framework to run time-insensitive applications as secondary background tasks but guarantee the stability characteristics. Cheng et al. [21] proposed an adaptive scheduling algorithm that schedules parallel micro-batch applications dynamically and automatically based on their data dependencies and workload properties. Chen et al. [22] designed a speculative slot reservation algorithm that reserves slots for upstream computations, which can also run extra copies of stragglers. However, all these do not take the time-critical feature of short applications into account.

Performance model. Wang et al. [23] presented a simulation driven prediction model to predict application performance for Apache Spark platform. OptEx [24] is another tool that models application completion time on Spark by considering the input dataset, the number of iterations, and the number of nodes. A dynamic memory manager for in-memory data analytics, MEMTUNE [25], is a dynamic memory manager that tunes computation/caching memory partitions at runtime based on memory demand and in-memory cache need. Yoo et al. [26] evaluated the performance and resource usage to decide when to scale-up and scale-out cluster for Spark. But none of these performance models is a comprehensive one that considers all performance-influence factors for Spark.

Multi-threading. Zhang et al. [27] designed a system called HJ-Hadoop to employ multi-core parallelism at JVM level to decrease the number of containers needed to run for Hadoop applications. Lion et al. [28] designed a new JVM that cuts long tasks into short ones to amortize JVM warm-up overhead through long tasks. These optimizations are only limited to tuning JVM level parallelism.

There are other aspects concerning distributed platform performance, such as network [29–31], HDFS [32], middleware [33, 34], and query optimization [35,36].

6. Conclusions and future work

Although big data platforms were originally designed for large-scale applications, the majority of these workloads are short. In this paper, we design and implement an optimized Spark-on-Yarn system for short applications by introducing three new operating modes: one-thread, one-container, and distributed. We introduce a brand-new scheduler that takes both data locality and resource usage into account and reduce the redundant communications between the Application Master and the Resource Manager. A new submitter framework is designed to avoid the setup time of Application Master and decide which mode is the most efficient based on our fine-grained performance model. According to our experimental results, our system is up to 2 times faster compared with the original Spark for short applications.

In the further, we plan to optimize our framework to decide the best mode with static program analysis and dynamic real-time analysis without history log information.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

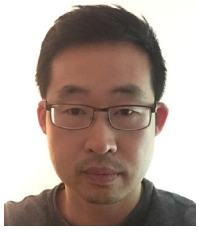
This work was supported in part by NSF, United States-1836881.

References

[1] Apache Spark website, http://Spark.apache.org/.
[2] S. Ryza, U. Laserson, S. Owen, J. Wills, Advanced Analytics with Spark: Patterns for Learning from Data at Scale, O'Reilly Media, 2015.
[3] Apache Hadoop website, http://hadoop.apache.org/.
[4] Apache Storm website, http://storm.apache.org/.
[5] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, in: OSDI 2004, 2004, pp. 1–13.
[6] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, I. Stoica, Improving mapreduce performance in heterogeneous environments, in: 8th USENIX Conference on Operating Systems Design and Implementation, 2008, pp. 1–14.
[7] K. Elmeleegy, Piranha: optimizing short jobs in hadoop, in: VLDB Endowment, 2013, pp. 985–996.
[8] H. Zhang, H. Huang, L. Wang, Mrapid: An efficient short job optimizer on Hadoop, in: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, IEEE, 2017, pp. 459–468.
[9] T. White, Hadoop: The definitive guide, "O'Reilly Media, Inc.", 2012.
[10] Apache Spring website, http://projects.spring.io/spring-hadoop/.
[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, I. Stoica, Dominant resource fairness: fair allocation of multiple resource types, in: NSDI, 2011, pp. 323–336.
[12] Asm website, http://asm.ow2.org/.
[13] M. Hammoud, M.F. Sakr, Locality-aware reduce task scheduling for mapreduce, in: CloudCom, 2011, pp. 570–576.
[14] X. Zhang, Z. Zhong, S. Feng, B. Tul, J. Fan, Improving data locality of mapreduce by scheduling in homogeneous computing environments, in: Int. Symp. on Parallel and Distributed Processing with Applications, 2011, pp. 120–126.
[15] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Maestro: Replica-aware map scheduling for mapreduce, in: CCGRID, 2012, pp. 435–442.
[16] V.S. Marco, B. Taylor, B. Porter, Z. Wang, Improving spark application throughput via memory aware task co-location: a mixture of experts approach, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, ACM, 2017, pp. 95–108.
[17] Y. Yao, J. Tai, B. Sheng, N. Mi, LsPS: A job size-based scheduler for efficient task assignments in hadoop, in: IEEE Transactions on Cloud Computing, 2014, pp. 411–424.
[18] J. Yan, X. Yang, R. Gu, C. Yuan, Y. Huang, Performance optimization for short mapreduce job execution in hadoop, in: CGC, 2012, pp. 1–7.
[19] F. Chen, S. Wu, H. Jin, Y. Yao, Z. Liu, L. Gu, Y. Zhou, Lever: towards low-latency batched stream processing by pre-scheduling, in: Proceedings of the 2017 Symposium on Cloud Computing, ACM, 2017, p. 643.

[20] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, T. Moscibroda, Tr-spark: Transient computing for big data analytics, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, ACM, 2016, pp. 484–496.
[21] D. Cheng, Y. Chen, X. Zhou, D. Gmach, D. Milojicic, Adaptive scheduling of parallel jobs in spark streaming, in: INFOCOM 2017-IEEE Conference on Computer Communications, IEEE, IEEE, 2017, pp. 1–9.
[22] C. Chen, W. Wang, B. Li, Speculative slot reservation: Enforcing service isolation for dependent data-parallel computations, in: Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on, IEEE, 2017, pp. 549–559.
[23] K. Wang, M.M.H. Khan, Performance prediction for apache spark platform, in: High Performance Computing and Communications (HPCC), IEEE, 2015, pp. 166–173.
[24] S. Sidhanta, W. Golab, S. Mukhopadhyay, Optex: A deadline-aware cost optimization model for spark, in: Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on, IEEE, 2016, pp. 193–202.
[25] L. Xu, M. Li, L. Zhang, A.R. Butt, Y. Wang, Z.Z. Hu, Memtune: Dynamic memory management for in-memory data analytic platforms, in: Parallel and Distributed Processing Symposium, 2016 IEEE International, IEEE, 2016, pp. 383–392.
[26] T. Yoo, M. Yim, I. Jeong, Y. Lee, S.-T. Chun, Performance evaluation of in-memory computing on scale-up and scale-out cluster, in: Ubiquitous and Future Networks (ICUFN), 2016 Eighth International Conference on, IEEE, 2016, pp. 456–461.
[27] Y. Zhang, Hj-hadoop: An optimized mapreduce runtime for multi-core systems, in: SPLASH, 2013, pp. 111–112.
[28] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, D. Yuan, Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems, in: OSDI, 2016, pp. 383–400.
[29] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, M. Caesar, Network-aware scheduling for data-parallel jobs: Plan when you can, in: ACM SIGCOMM, 2015, pp. 407–420.
[30] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, P. Chen, A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus, in: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, ACM, 2011, p. 2.
[31] L. Wang, S. Lu, X. Fei, A. Chebotko, H.V. Bryant, J.L. Ram, Atomicity and provenance support for pipelined scientific workflows, Future Gener. Comput. Syst. 25 (5) (2009) 568–576.
[32] H. Zhang, L. Wang, H. Huang, Smarth: Enabling multi-pipeline data transfer in hdfs, in: ICPP, 2014, pp. 1–10.
[33] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, H. Bal, Ajira: A lightweight distributed middleware for mapreduce and stream processing, in: ICDCS, 2014, pp. 545–554.
[34] Z. Liu, H. Zhang, L. Wang, Hierarchical spark: A multi-cluster big data computing framework, in: Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on, IEEE, 2017, pp. 90–97.
[35] H. Zhang, Z. Sun, Z. Liu, C. Xu, L. Wang, Dart: A geographic information system on hadoop, in: IEEE CLOUD, IEEE, 2015, pp. 1–8.
[36] H. Huang, J.M. Dennis, L. Wang, P. Chen, A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography, Procedia Comput. Sci. 18 (2013) 581–590.

Hong Zhang received the M.Sc. degree in the Department of Computer Science from University of Wyoming, Laramie, WY, in 2015. He is currently working towards the Ph.D. degree with the Department of Computer Science, University of Central Florida. His research interests include high performance computing, performance evaluation, and optimization of cloud and big data service such as Hadoop, and Spark. He is a student member of the IEEE.



Dr. Hai Huang is a Research Staff in the Cloud Platform division at IBM Research. His research interests include operating systems, energy and power management, large-scale systems management, software testing, anomaly detection, and new systems challenges in the field of Cloud Computing. Prior to joining IBM Research, he received his Ph.D. in Computer Science and Engineering (CSE) from University of Michigan in 2006. He received his MS in CSE from University of Michigan in 2002 and BSE in CSE from the Ohio State University in 2000. He has over 30 peer reviewed papers in conferences such as SOSP, USENIX ATC/FAST, and EUROSYS, and filed over 20 patents.

Liqiang Wang received the Ph.D. degree in Computer Science from Stony Brook University in 2006. He is an Associate Professor in the Department of Computer Science at the University of Central Florida. His research interest is the design and analysis of parallel systems for big-data computing, which includes two aspects: design and analysis. For design, he is currently working on optimizing performance, scalability, resilience, load balancing of data-intensive computing and distributed machine learning. For the aspect of analysis, he focuses on using program analysis to detect programming errors and performance defects in large-scale parallel computing systems. He received an NSF CAREER Award in 2011.