2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)

A Model Driven Approach towards Improving the Performance of Apache Spark Applications

Kewen Wang, Mohammad Maifi Hasan Khan, Nhan Nguyen and Swapna Gokhale
Department of Computer Science and Engineering
University of Connecticut
kewen.wang@uconn.edu, maifi.khan@uconn.edu,nhan.q.nguyen@uconn.edu,ssg@engr.uconn.edu

***Abstract*—Apache Spark applications often execute in multiple stages where each stage consists of multiple tasks running in parallel. However, prior efforts noted that the execution time of different tasks within a stage can vary significantly for various reasons (e.g., inefficient partition of input data), and tasks can be distributed unevenly across worker nodes for different reasons (e.g., data co-locality). While these problems are well-known, it is nontrivial to predict and address them effectively. In this paper we present an analytical model driven approach that can predict the possibility of such problems by executing an application with a limited amount of input data and recommend ways to address the identified problems by repartitioning input data (in case of task straggler problem) and/or changing the locality configuration setting (in case of skewed task distribution problem). The novelty of our approach lies in automatically predicting the potential problems a priori based on limited execution data and recommending the locality setting and partition number. Our experimental result using 9 Apache Spark applications on two different clusters shows that our model driven approach can predict these problems with high accuracy and improve the performance by up to 71%.**

***Index Terms*—Apache Spark; Task Imbalance; Straggler; Performance Modeling; Performance Optimization; Task Distribution; Configuration Tuning;**

I. Introduction

Apache Spark is a recently popularized data analytic platform that is being adopted by numerous organizations (e.g.,Yahoo!, eBay, Baidu, Netflix [1]) due to its in-memory computation framework and superior performance [2].

On this platform, each application is executed in multiple stages where each stage can execute multiple tasks in parallel. While this model helps to speed up the execution significantly, this is shown to lead to two different problems as follows [3]–[11]. *First,* among the tasks within a stage, occasionally some tasks may take much longer (i.e., task stragglers) than the median task time due to suboptimal partition of input data and can increase the job completion time significantly. *Second,* as Apache Spark tries to assign computing tasks closer to its input data to reduce execution time, this can lead to skewed task distribution across worker nodes, resulting in wasted resources and increase in job completion time. While Apache Spark provides locality setting (i.e., spark.locality.wait) to control the process of task creation on remote worker nodes to address possible skewed task distribution problem, it is nontrivial to tune this setting.

Prior efforts attempted various approaches such as redesigning task scheduler [6], [8], [9] and designing locality manager to address these problems [7]. However, as each application is different and even the same application can perform differently based on input data characteristics, it is extremely difficult to have a generalized algorithm that works for each application. Furthermore, while intelligent job scheduling may solve the task straggler problem caused by inefficient scheduling, it may not mitigate the task straggler problem caused by data skew or other reasons.

To address this problem, in this paper we take a model driven approach where a given application is first run with a fraction of the input data set to predict possible stragglers and/or skewed task distribution problem in advance. Subsequently, if the model predicts the possibility of task straggler problem, we use our performance models to repartition the input data (or intermediate data if needed) by adjusting the partition number to either split a longer task (i.e., straggler) into multiple shorter tasks, or merge multiple tiny tasks into a larger one. On the other hand, if the model predicts the possibility of skewed task distribution problem, we tune the locality setting (i.e., spark.locality.wait) that controls the task creation on remote worker nodes to address possible skewed task distribution problem (details are explained in Section III).

Our experimental results using 9 Apache Spark applications on two different clusters show that our model driven approach can mitigate the predicted imbalance problem and improve performance significantly.

II. Related Work

Prior effort exists that looked into various aspects of cloud platform performance troubleshooting and tuning such as addressing task straggler problem, load balancing, and resource allocation [6]–[9], [11]–[22]. Among these, Sparrow [6] is implemented on top of Spark that aims to balance task load across nodes through sub-second task scheduling. Sparrow's performance is evaluated using TPC-H query benchmark, and is shown to reduce the median query response time by 4-8X. Stark [7] in contrast presents a locality manager that enforces data co-locality to reduce shuffle cost, and uses group manager to dynamically change partition number to balance task execution time. This tool is shown to reduce job execution time by 4X compared to Spark (version 1.3.1) in case of log mining jobs. Ernest [22] focuses on predicting run time for a target hardware configuration (e.g., number of nodes), and uses sampling and training method to build application performance

978-1-7281-0746-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ISPASS.2019.00036

IEEE computer society

models. However, it is not designed to identify the underlying root cause (e.g., improper locality setting, suboptimal partition of data) or address them by manipulating software configuration. Rather, it attempts to address the problem by allocating additional hardware resources.

Researchers have looked into job scheduling and resource allocation for other platforms such as Hadoop [23] as well [12]–[17], [21], [24], [25]. Among these, Wrangler [12] changes the fair scheduler of Hadoop, and schedules tasks based on the prediction of whether an incoming task will be a straggler or not. It is shown to improve job completion time by 61% and 43% at 99% percentile for two types of real world workloads. Dolly [13] on the other hand tries to eliminate task straggler problem through full cloning of small jobs ($\leqslant$10 tasks), and uses delayed assignment to avoid the contention.

There are other approaches that focus on efficient load balancing techniques [14], [21], [24] in order to improve performance. For instance, Libra [14] uses sampling methods to collect partial map task information to estimate the distribution of intermediate data, and uses range partition to evenly partition data. It is shown to achieve up to 4X improvement in terms of job execution time.

While many of these prior efforts focus on addressing task straggler problem, in contrast to prior efforts, we focus on developing analytical models for predicting the possibility of task straggler and skewed task distribution problem. As such, our work is complementary to prior approaches, and helps users to understand the underlying reasons behind suboptimal performance problem and address them by modifying a target application with minimal effort.

III. Overview

Apache Spark applications are executed in multiple stages where each stage contains multiple tasks running on multiple worker nodes in parallel. As the later stages of an application often cannot start until all the tasks of the previous stage are finished, stragglers within a stage can heavily impact the subsequent stages and the overall application completion time. Furthermore, if tasks are assigned to different nodes disproportionately, nodes running more tasks may need longer time to complete compared to nodes running fewer tasks, negatively affecting the application completion time.

Instead of trying to detect and address such problems during runtime, we try to predict these two categories of problems (i.e., task straggler and skewed task distribution) a priori by running the target application with only a fraction of the input data as shown in Figure 1. Subsequently, based on the performance model, we identify the stage(s) where data may need to be repartitioned by modifying the corresponding part of the source code (in case of stragglers) and/or change the configuration setting to improve task distribution (in case of skewed task distribution) to address the problems. The details are below.

A. Performance Model for Apache Spark Application

Given the multistage execution model where different stages within an application may be executed either sequentially or in parallel (if possible), we represent the execution time of an application as the sum of the execution time of the stages plus the application startup time and cleanup time as follows.

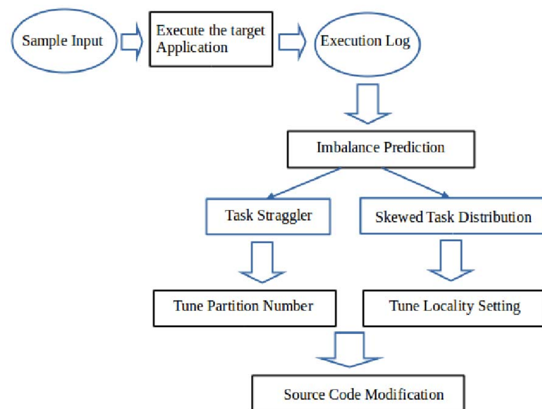


Fig. 1: Workflow of the Presented Approach

$$AppTime = Startup + StagesTime + Cleanup \quad (1)$$

$$StagesTime = \sum_{i=1}^{N_s} \max_{j=1}^{N_i} StageTime_{i,j} \quad (2)$$

Here, N_s is the number of sequential segments of stages, and N_i is the number of parallel stages within each segment of stages. Note that the number of tasks in each $Stage_i$ may not be equal as some stages may run in parallel. For example, if two stages run in parallel, the following stage will contain twice as many tasks as the number of tasks in the previous stage.

In a cluster of H worker nodes, the number of total CPU cores P can be calculated as follows.

$$P = \sum_{i=1}^{H} Core_i \quad (3)$$

Here, $Core_i$ is the number of CPU cores of worker node i. As one CPU core executes one task at a time, P is the maximum number of tasks that can be run in parallel. Hence, within an execution stage, tasks in each stage are executed in batches where each batch consists of P tasks running in parallel. Therefore, the execution time of a stage can be calculated as the maximum of the sum of all the sequential tasks' time within a stage plus the stage startup time and the cleanup time as follows.

$$StageTime = Startup + \max_{h=1}^{H} \max_{c=1}^{Core_h} \sum_{i=1}^{N_c} TaskTime_{c,i} + Cleanup \quad (4)$$

Here $Core_h$ is the number of CPU cores in host h and N_c is the number of tasks executed on CPU core c.

As different tasks in a stage follow the same execution pattern, the execution time of a task can be computed as follows.

$$TaskTime_i = SchedulerDelay_i + DeserializationTime_i + RunTime_i + SerializationTime_i \quad (5)$$

Here $SchedulerDelay_i$ is the task scheduling delay, $DeserializationTime_i$ is the time taken to deserialize a task object, $SerializationTime_i$ is the time taken to serialize the

(intermediate) result, and $RunTime_i$ is the time spent in performing operations such as data mapping, reduction, and sorting.

B. Model based Prediction of Potential Imbalance Problems

The main idea behind our work is to execute an application with a smaller input data set, which is sampled from the actual target data set, and analyze the execution profile to predict potential imbalance problems for the application.

1) Predicting the Problem of Task Straggler:

In an ideal case, for a given application, the execution time T of each of the N tasks within a stage should be close to each other. As such, within a stage, if a significant difference exists among task execution time $T_i(1 \leqslant i \leqslant N)$, it indicates the possibility of stragglers. From our experiments, we noted that the average execution time of the first wave of tasks is different from the average execution time of the following waves of tasks. To account for this phenomenon, we calculate the median execution time T_m for the tasks in the first wave and the following waves separately.

In this work we define a ratio RS as shown in equation (6), where if $RS \geqslant \alpha$ as in equation (7) we decide that the task straggler problem exists for this application.

$$RS = \max_{i=1}^{N} T_i / T_m \tag{6}$$

$$RS \geqslant \alpha, \quad \alpha = 2 \tag{7}$$

In this work we choose $\alpha = 2$ to identify tasks that take significantly longer to execute compared to other tasks. Note that this value is determined empirically and based on prior works [3], which can be changed to adjust the sensitivity of the prediction algorithm.

2) Predicting the Problem of Skewed Task Distribution:

We identify two possible reasons that may cause skewed task distribution problem as follows.

To identify skewed task distribution caused by improper locality setting, first, we analyze the application execution logs and identify the number of tasks N_i launched on each node $i(1 \leqslant i \leqslant H)$.

Next, we define a task distribution ratio RD as shown in equation (8). If $RD \geqslant 2$ and the median execution time $T_m \leqslant 2s$, we predict that skewed task distribution problem exists for this application as shown in equation (9). Intuitively, equation (9) identifies whether a node has at least twice as many tasks as other nodes ($RD \geqslant 2$) and $T_m \leqslant 2s$, implying a large number of small tasks are scheduled on one node (due to improper locality setting).

As the default waiting period for a task before it is scheduled on a lower locality level node is 3s, we select 2s as the threshold in our model, which was determined empirically. Furthermore, as the total number of tasks in the simulated execution might be smaller than the total number of CPU cores in the cluster, it is possible that some working nodes may have no task to run. For that, we only calculate RD for working nodes that have at least one running task. If $RD < 2$ and the number of tasks $\geqslant H$, we predict the possibility of potential skewed task distribution problem as in equation (10). Here N_{idle} is the number of idle nodes that have no task to run. Intuitively, this condition checks whether there are idle cores while some node has multiple small tasks ($T_m \leqslant 2s$) (due to improper locality setting).

$$RD = \max_{i=1}^{H} N_i / \min_{i=1}^{H} N_i \tag{8}$$

$$RD \geqslant \beta \quad and \quad T_m \leqslant 2s, \quad \beta = 2 \tag{9}$$

$$N_{idle} > 0, \sum_{h=1}^{H} N_h \geqslant H, RD < 2, T_m \leqslant 2s \tag{10}$$

In addition to the improper locality setting, we identify that suboptimal partition number may lead to the creation of a small number of tasks, leading to cluster underutilization and skewed task distribution as well. For instance, in a cluster of H working nodes, creating less than H tasks will lead to at least one idle node, causing skewed task distribution as well. This is identified using the following equation.

$$\sum_{h=1}^{H} N_h < H \tag{11}$$

C. Addressing the Predicted Problem

Once we predict the problem (i.e., task straggler and/or skewed task distribution problem) as explained in the previous section, the next step is to address the problem as follows.

1) Addressing the Problem of Task Stragglers:

In Apache Spark the input data is evenly partitioned among all the tasks in the first stage. However, this does not guarantee that the key distribution will be even once the input data is mapped onto key-value pairs and may have large variance, causing tasks responsible for processing extra key-value pairs taking longer to finish compared to tasks with fewer key-value pairs. This often results in significant differences in the time needed to perform the shuffling operation as the key-value pairs sharing the same key range need to be fetched together for subsequent processing.

Furthermore, during the execution of multiple stages, the size of output data of an intermediate stage is likely to change, causing the size of the input data for the following stages to change as well. Such runtime variations may cause the data partition used in the previous stage to become suboptimal for the following stages. In the worst case, some partitions may contain no data at all although one task will be assigned to each such partition, causing skewed task time distribution.

As task stragglers often result from inappropriate partitioning of data, one way to solve this is by repartitioning data that can divide the long tasks into multiple smaller tasks and execute on multiple nodes. Repartitioning can also eliminate the cost of creating a large number of tiny tasks. While re-partitioning can be done using "repartition()" operation on Apache Spark platform, however, this operation is I/O intensive as it involves remote data shuffling. Instead, we attempt to change the partition number for the stage where the straggler is predicted.

To address this, we develop performance models to estimate the execution time for a target application for a given partition number and use this model to find out a possible partition number that can reduce the application execution time and

eliminate possible task straggler problem. Note that, while the suggested partition number by our algorithm is likely to improve performance significantly, it may not be optimal which will require testing all possible values and is computationally infeasible.

We identify the stragglers using the following equation.

$$ImbTasks = \{Task_i \mid Time_{Task_i} \geqslant \alpha Time_m\} \quad (12)$$

Here, $\alpha = 2, 1 \leqslant i \leqslant N$, and $Time_m$ is the median value of the task execution time. As the average execution time of the tasks in the first wave is significantly different compared to the subsequent waves within the same stage, we calculate the median value for task execution time in the first wave and the following waves separately.

Once we change the partition number, the number of initial tasks will be changed as well. We calculate the ratio γ as in equation (13).

$$\gamma = N_t/N \quad (13)$$

Here N_t is the number of estimated new tasks, and N is the number of current tasks. When N_t becomes larger than N, then γ will be greater than 1 and the task execution time will be shorter. Once the partition number is changed, the execution time for each task is changed as well.

For non-straggler tasks, we use the average task time in each host h to estimate the new average task time as follows.

$$EstAvgTaskTime = \frac{\sum_{i=1}^{N-N_u}(\frac{RunTime_i}{\gamma} + MTime_i)}{N - N_u} \quad (14)$$

$$MTime_i = TaskTime_i - RunTime_i \quad (15)$$

Here N is the number of total tasks, and N_u is the number of straggler tasks. $RunTime_i$ is the time spent performing operations such as map, $sort$, $reduce$ in $Task_i$. $MTime_i$ includes time spent on scheduling and/or serialization.

We estimate the task time for straggler tasks when the partition number is changed as follows.

$$EstImbTaskTime_k = RunTime_k/\gamma + ImbTaskTime_k - RunTime_k, 1 \leqslant k \leqslant N_u \quad (16)$$

Here $ImbTaskTime_k$ refers to the straggler task execution time before the partition number is changed.

Next, we estimate the new task time $NewImbTaskTime_i$ for straggler tasks when the partition number is changed as follows. When $\gamma > 1$, as there will be more straggler tasks, we use the execution time of the original set of straggler tasks to estimate the execution time of the new set of straggler tasks. In contrast, when $\gamma < 1$, as there will be fewer straggler tasks, we use the average time of the original set of straggler tasks to estimate the new execution time.

$$NewImbTaskTime_i = \begin{cases} EstImbTaskTime_{i \,\%\, N_u}, & \gamma \geqslant 1 \\ \sum_{k=1}^{N_u} \frac{EstImbTaskTime_k}{N_u}, & \gamma < 1 \end{cases}$$
$$1 \leqslant i \leqslant \lceil \gamma \times N_u \rceil \quad (17)$$

Next, we use the estimated task execution time to assign non-straggler tasks and straggler tasks to available cores wave by wave and calculate the execution time of the modified stages and update the application time accordingly.

Using the above formulation, we use Algorithm 1 to search for the partition number that may reduce the application execution time the most. Briefly, the algorithm begins with partition number P, and increases the partition number by P in each iteration until it reaches the maximum possible value. In each iteration, it estimates the application execution time for the new partition number using Algorithm 2. At the end, the algorithm outputs the partition number that is predicted to achieve the shortest execution time. As we use our model to predict the execution time in each iteration, the runtime of the algorithm is less than 5s for all applications in our experiment, and is less than 1s in most cases.

```
    Input: SampleApp
    Output: Recommended Partition Number bpn
 1  Function PartitionNumSearch
 2      Current PartitionNum cpn;
 3      P = sum_{i=1}^{H} Core_i;
 4      Initialize PartitionNum pn = P, δ = P;
 5      Initialize mpn = Max{16 × cpn, 100 × P};
 6      Initialize RecAppTime=MaxValue;
 7      while pn ⩽ mpn do
 8          ratio γ = pn/cpn;
 9          AppTime = EstTime(SampleApp, γ);
10          if AppTime − RecAppTime < −1s then
11              RecAppTime = AppTime;
12              bpn = pn;
13          end
14          pn+ = δ
15      end
16  end
```

Algorithm 1: PartitionNumber Search Algorithm

```
    Input: SampleApp, γ
    Output: Estimated AppTime AppTime
17  Function EstTime
18      Initialize startID = Stage.ID of the first stage;
19      for Stage ∈ SampleApp.Stages do
20          if Stage.ID < startID then
21              γ = 1;
22          end
23          Estimate non-straggler task time using eq (14);
24          Estimate straggler task time using eq (17);
25          Estimate stage time StageTime;
26          if ∃ Parallel Stages then
27              AppTime += max_{i=1}^{S} StageTime_i; here S is
                the number of parallel stages
28          else
29              AppTime += StageTime;
30          end
31      end
32  end
```

Algorithm 2: Algorithm for Application Time Estimation

TABLE I: Data Locality Level

Locality Level	Meaning
PROCESS_LOCAL	On the same JVM
NODE_LOCAL	On the same node
NO_PREF	No locality preference
RACK_LOCAL	On the same rack of servers
ANY	Not in the same rack

TABLE II: Locality Configuration

Configuration	Default Value	Meaning
spark.locality.wait	3s	Wait time for node in All locality levels
spark.locality.wait.process	spark.locality.wait	Wait time for node in PROCESS_LOCAL
spark.locality.wait.node	spark.locality.wait	Wait time for node in NODE_LOCAL
spark.locality.wait.rack	spark.locality.wait	Wait time for node in RACK_LOCAL

TABLE III: Operations with Partition Parameters

Function Name	Partition Parameters		Applications
	partitioner	numPartitions	
aggregateByKey	✓	✓	
coalesce		✓	
combineByKey	✓	✓	
distinct		✓	
edgeListFile		✓	TriangleCount ConnectedComponents
groupByKey	✓	✓	PageRank
intersection	✓	✓	
join	✓	✓	
logNormalGraph		✓	SSSP
reduceByKey	✓	✓	
repartition		✓	
sortByKey		✓	

2) Addressing the Problem of Skewed Task Distribution: To reduce execution time, Apache Spark tries to assign computing tasks closer to its input data. Based on distance, it has multiple locality levels as listed in Table I. The best case scenario is when Apache Spark is able to schedule a task on the same JVM where the input data for that task is located. However, if the node where the input data is located is overloaded, it is possible that a task may have to wait a long time before it can be executed. To avoid such long waiting time, Apache Spark provides a set of configuration parameters (listed in Table II) that can be set to control the maximum waiting period before a task is scheduled on a lower locality level. For example, in case of the default setting, it will wait 3 seconds before it schedules a task on a remote node. As a result, for small tasks with run time less than 3 seconds, a large number of short tasks may be accumulated in one or a small number of nodes while some nodes may remain idle.

To address the problem of *skewed task distribution*, we use our analytical performance models to estimate the application execution time considering tasks were assigned evenly across nodes. If the estimated application execution time is less than the original execution time, we update the locality configuration setting.

In case of skewed task distribution due to cluster under-utilization, we use Algorithm 1 described in Section III-C to search for a partition number that will launch additional tasks to effectively utilize the cluster. Based on the estimated difference in execution time, the algorithm determines whether to change the partition number or not.

D. Source Code Modification to Tune Locality Setting and Partition Number

Once the algorithm identifies the recommended partition number and/or the locality setting, we need to modify the Spark application source code to see the actual effect. Note that this change is minimal and often involves only changing one to two lines of code as follows.

To change the locality setting (which is referred to as "Improvement 1" in the paper), we modify the application source code by adding "$.config(spark.locality.wait, 0)$" to change the waiting period before creating tasks on non-local nodes to balance the task distribution.

To change the partition number (which is referred to as "Improvement 2" in the paper), we first identify the application stage(s) and the corresponding operations (e.g., reduce(), distinct(), join()) associated with the task straggler problem by analyzing the execution log. For example, if we identify that task stragglers exist in the second stage, we change the partition number for the operation that corresponds to the second stage.

One way to change the number of partitions for the following stages is by inserting operations such as "repartition(n)" before the target stage. However, as using this operation will add more stages and may increase the execution time, instead, we change the partition number parameter of the corresponding operation for that stage. As some operations already contain the partitioning parameter, we take advantage of this feature to perform the repartitioning without adding extra stage to the application. Among the Apache Spark operations listed in Table III, some contain parameter that can directly change the partition number such as "$groupByKey$." Note that Table III does not list all the functions that allow changing the partition number. In addition to that, some functions of Spark libraries also provide partition number parameter such as function "$edgeListFile$" used in the Spark graphx library. We can use this feature to change the partition number for graph processing applications that use Spark graphx library (e.g., TriangleCount, ConnectedComponents).

In cases where no such function is used in the straggler stage, we can change the partition number in an earlier stage, which impacts the number of partitions for the subsequent stages.

Finally, some applications use input functions to obtain the input data. These input functions (listed in Table IV) can either use parameter partition number to change the number of partitions directly or use other parameters to determine the minimum value for partition number (e.g., "$textFile$").

IV. Evaluation

To evaluate the effectiveness of our approach, we used two sets of clusters (Table V). $ClusterI$ consists of 6 nodes where one node was used as the master node and 5 nodes as worker

TABLE IV: Input Functions with Partition Parameters

Function Name	Partition Parameters		Applications
	numPartitions	minPartitions	
binaryFiles		✓	
hadoopRDD		✓	
makeRDD	✓		
parallelize	✓		PiEstimation
range	✓		
sequenceFile		✓	
textFile		✓	TransitiveClosure WordCount

TABLE V: Cluster Setting

Physical Configuration	Cluster I	Cluster II
Number of Nodes	6	4
Number of CPU cores per Node	8	20
Memory size per Node	22GB	16GB

TABLE VI: Apache Spark Applications

Application	Imbalance	Sample Input	Input
TransitiveClosure	✓	1GB	10GB
TriangleCount	✓	2GB	50GB
PageRank	✓	2GB	50GB
ConnectedComponents	✓	2GB	50GB
SSSP	✓	10000 vertex	1000000 vertex
PiEstimation	✓	1000 slices	100000 slices
WordCount	✓	2GB	20GB
K-Means		2GB	20GB
LogisticRegression		2GB	20GB

nodes. In this cluster, each machine is configured with 8 CPU cores and 22GB of RAM memory. $ClusterII$ consists of 4 nodes where one node was used as the master node and 3 nodes as worker nodes. In this cluster, each machine is configured with 20 CPU cores and 16GB of RAM memory. We ran Apache Spark (version 2.1.0) on top of Apache Hadoop Distributed File System (version 2.7.3), and used the standalone mode for Apache Spark platform. The default block size of Apache Hadoop Distributed File System (HDFS) is set at 128MB which affects the number of partitions and task number. We used the Apache Spark log files to analyze the execution data for each application.

In order to evaluate the performance of our approach, we used 9 different Apache Spark applications as listed in Table VI. TriangleCount, PageRank, and ConnectedComponents were tested using the LiveJournal network dataset downloaded from SNAP [26]. The input data is pre-processed by mapping each node id onto a longer string to form a 50GB dataset. TransitiveClosure was tested using 10GB data sampled from this 50GB data set as the system ran out of memory when 50GB input data was used. SSSP (SingleSourceShortestPath) used generated graph with 1000000 vertices, while PiEstimation used parallelized number collection of 100000 slices. WordCount application was tested using 20GB Wikipedia dump data. Lastly, K-Means and LogisticRegression were tested using 20GB of numerical Color-Magnitude Diagram data downloaded from Sloan Digital Sky Survey (SDSS) [27].

To predict the potential imbalance problems for these 9 applications, we first executed each application with sample input data, which was randomly extracted from the complete input data set, and then analyzed the execution profile for each sample application to predict potential task straggler and/or skewed task distribution problem based on equation (7), (9), (10) and (11).

Note that the maximum size of sample input data was 2GB for 50GB input size (4%). While we intended to test all applications with 50GB input data, however, due to limited scale of our cluster we experienced runtime errors for some applications when executed with 50GB dataset (e.g, TransitiveClosure). In such cases, we had to reduce the input size to 10GB, causing the sample to input data ratio to become 10%. Table VI lists the size of sample dataset for all applications. The block of sample dataset was selected from a randomly selected position of the whole dataset.

We calculated the runtime overhead of our approach as a percentage of total execution time before improvement. The overheads (including sampling, profiling and searching for best configuration) were 5%, 15%, 12%, 35%, 9%, 6%, and 16% for TriangleCount, PageRank, ConnectedComponents, WordCount, TransitiveClosure, SSSP, and PiEstimation respectively. As jobs on production systems are likely to run for much longer (i.e., hours or days) compared to our limited scale evaluation setup, we expect the relative overhead to be much smaller in real deployment.

To evaluate the effect of addressing potential skewed task distribution problem, we use equation (18) to calculate the standard deviation for the number of tasks across all the working nodes in the cluster as follows.

$$std = \sqrt{\sum_{h=1}^{H} (N_h - \bar{N})^2/(H-1)} \qquad (18)$$

Here N_h refers to the number of tasks on node h.

Table VII displays the prediction result of our algorithm. "True Positive(TP)" indicates that the listed problem exists for a given application and is correctly identified by our algorithm. "False Negative(FN)" in this table indicates that the algorithm incorrectly predicts that there will be no imbalance problem although the problem exists.

Out of the 9 applications, 7 of them were correctly predicted to have either task straggler and/or skewed task distribution problem on $ClusterI$, while 6 of them were predicted to have either task stragglers and/or skewed task distribution problem on $ClusterII$. Note that due to the differences in CPU and memory capability between these two clusters, the imbalance problem disappeared or intensified in some cases depending on the cluster. For instance, PiEstimation exhibited task straggler problem on $ClusterI$ where the ratio RS was larger than 20 whereas this ratio RS dropped to around 2 on $ClusterII$. As such, evaluation on two different clusters was performed to test the effectiveness of our approach.

Based on our evaluation, we classified these 9 applications into three groups. The first group included applications that were predicted to have both stragglers and skewed task distribution problem. The second group included applications that were predicted to have either task straggler problem or skewed task distribution problem. Finally, the last group included applications that were predicted to have no imbalance problem.

Table VIII and Table IX list the performance improvement after addressing the predicted problems using our approach. Here, column "Improvement 1" lists the performance im-

TABLE VII: Imbalance Prediction Result

Application	Straggler		Skewed Distribution	
	Cluster I	Cluster II	Cluster I	Cluster II
TriangleCount	TP	TP	TP	TP
PageRank	TP	TP	TP	TP
ConnectedComponents	TP	TP	TP	FP
WordCount	TP	TP	TP	FP
TransitiveClosure	TP	TP	FN	FN
SSSP	TN	TN	TP	TP
PiEstimation	TP	FN	TN	TN
K-Means	FN	FN	FN	TN
LogisticRegression	FN	TN	TN	TN

provement for applications by tuning locality setting, column "Improvement 2" lists the performance improvement for applications by tuning partition number, and column "Improvement 1&2" lists the performance improvement for applications by tuning locality setting and partition number. As $ClusterII$ had more CPU cores and few working nodes, the imbalance problem was often less severe, causing the performance improvement to be lower on $ClusterII$ in some cases. In addition, we list the performance with the speculative execution feature enabled, which is a built-in approach for mitigating the task straggler problem provided by the Apache Spark platform. As can be seen in the table, compared to speculative task execution, which negatively impacted performance in some cases, our approach performed significantly better and improved performance by up to 71%. Details are below.

A. Group I: Applications with both task straggler and skewed task distribution problems

TriangleCount, PageRank, ConnectedComponents and WordCount exhibited both task straggler and skewed task distribution problems and were assigned to this group. To address the skewed task distribution problem, we first changed the locality configuration "spark.locality.wait" to 0 seconds (Improvement 1). Next, to address the task straggler problem, we attempted to find a better partition number (Improvement 2). We list the application execution time with enabling the speculative task execution (Config Speculation) feature of Apache Spark as well. Figure 2a and Figure 2b display the performance improvement for Group I applications for each cluster respectively. As TriangleCount ran out of memory on $ClusterII$ when $50GB$ input data was used, the input data for TriangleCount was changed to 10GB, resulting in much shorter execution time for TriangleCount on $ClusterII$ compared to $ClusterI$.

For TriangleCount application on $ClusterI$, we set the partition number to 40 as suggested by our algorithm (the default value was 400), which improved the performance by 63% compared to the default execution time (Table VIII). Moreover, Figure 3a and Figure 4a show the values for RS and std for task number in each stage of the application before and after improvement. While RS was increased after "Improvement 1", it dropped to a lower level after both improvement methods were applied. It may be due to the fact that the purpose of "Improvement 1" is to balance the task distribution and not to balance the task execution time. "Improvement 2" can reduce RS compared to "Improvement 1." Applying "Improvement 1&2" achieved better performance for task distribution compared to "Improvement 1" alone as shown in Figure 4a.

For PageRank application on $ClusterI$, it achieved 40% improvement after we addressed the skewed task distribution problem (Improvement 1). However, when we selected the partition number suggested by our algorithm, the execution time was not improved further. Enabling the speculative execution feature improved the performance of PageRank by 25%, which was the highest among these four applications. Figure 3b and Figure 4b display RS and std for task number in each stage of the application before and after improvement. The reduction in std for task number after the improvement can be seen in Figure 4b. However, RS was increased after the improvement method was applied as the task straggler problem mostly resulted due to the scheduling issue instead of inappropriate partitioning.

For ConnectedComponents on $ClusterI$, addressing the skewed task distribution problem improved the performance by approximately 10%. It was then improved by about 32% after using the suggested partition number. Compared to TriangleCount, as ConnectedComponents application contained a large number of short tasks ($time < 1s$), we had to reduce the partition number. As can be seen in Table VIII, using partition number 40 achieved 32% performance improvement compared to using the default partition number 400. Figure 3c and Figure 4c show RS and std for task number in each stage of the application before and after improvement.

For WordCount application on $ClusterI$, the performance improvement was small (around 4%) as it had no significant task imbalance problem originally. Also, our algorithm suggested 160 as the partition number, which was the default partition number for this application. Figure 3d and Figure 4d show RS and std for task number in each stage of the application before and after improvement. The execution time of the first 3 stages contributed significantly to the total execution time, and the RS for these first 3 stages were small, indicating that this application had no serious task straggler problem, which explains why "Improvement 2" helped little to improve the performance.

B. Group II: Applications with one imbalance problem

TransitiveClosure, SSSP and PiEstimation belonged to this group. Figure 2c and Figure 2d show the performance improvement for Group II applications on each cluster. PiEstimation is not included in Figure 2d as it was predicted to have no imbalance problem when run on $ClusterII$. For TransitiveClosure application, most of the tasks had execution time longer than 3s. For SSSP, the partition number was always 2 under default setting, which led to skewed task distribution problem. For PiEstimation, there were a large number of tasks with execution time shorter than $200ms$.

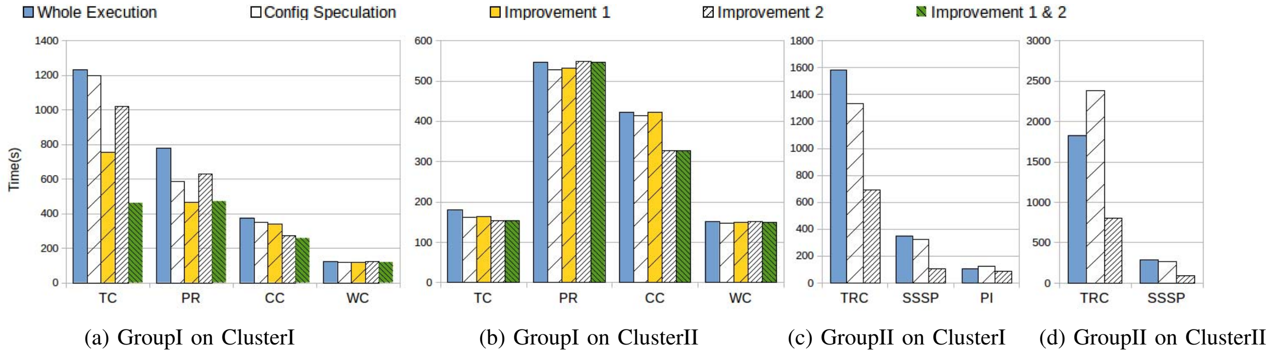
For TransitiveClosure application on $ClusterI$, by analyzing the sample execution log we found a task that took 30 times longer than the median task execution time in a

TABLE VIII: Performance Improvement on Cluster I

Application	Abbrev.	Config Speculation	Improvement 1 (Tune Locality Setting)	Improvement 2 (Tune Partition Number)	Improvement 1&2
TriangleCount	TC	3%	39%	17%	63%
PageRank	PR	25%	40%	19%	39%
ConnectedComponents	CC	7%	10%	27%	32%
WordCount	WC	3%	4%	0%	4%
TransitiveClosure	TRC	16%	-	56%	-
SSSP	SSSP	8%	-	71%	-
PiEstimation	PI	-22%	-	14%	-

TABLE IX: Performance Improvement on Cluster II

Application	Abbrev.	Config Speculation	Improvement 1 (Tune Locality Setting)	Improvement 2 (Tune Partition Number)	Improvement 1&2
TriangleCount	TC	10%	9%	15%	15%
PageRank	PR	3%	3%	0%	0%
ConnectedComponents	CC	2%	0%	22%	23%
WordCount	WC	2%	2%	0%	2%
TransitiveClosure	TRC	-30%	-	56%	-
SSSP	SSSP	8%	-	68%	-



(a) GroupI on ClusterI (b) GroupI on ClusterII (c) GroupII on ClusterI (d) GroupII on ClusterII

Fig. 2: Performance Improvement for Apache Spark Applications

particular stage for the default partition number of 80 for 10GB input data. From Table VIII, we can see that the performance was improved by 56% after changing the partition number to 240 as suggested by our algorithm. Figure 3e and Figure 4e show RS and std for task number in each stage of the application before and after improvement. As can be seen, RS was reduced after the improvement, especially in stage 3 where the longest task was launched. As such, mitigating the straggler problem in stage 3 significantly improved the application performance.

For SSSP (SingleSourceShortestPath) application on $ClusterI$, it used a default value of 2 for partition number. From Table VIII, we can see that using partition number 40 improved the performance by 71%. Figure 3f and Figure 4f show RS and std for task number in each stage of this application before and after improvement. Improvement II resulted in more tasks and increased the std for task number. However, tasks were more evenly distributed across nodes, thereby improving performance.

PiEstimation used the number of slices as the number of partition number. When we used 1000 slices as input for this application in our sample execution, it contained 1000 partitions. As a result, it contained a large number of tasks with execution time shorter than $200ms$. In contrast, when we used 100000 slices as input, our algorithm identified 7880 as the recommended partition number, which outperformed 100000. From Table VIII, we can see that using partition number 7880 achieved 14% performance improvement.

As PiEstimation had 1000 tasks in the sample execution, RD was used to predict the skewed task distribution problem, and no skewed task distribution problem was predicted. Figure 3g and Figure 4g show RS and std for task number in each stage of this application before and after improvement. As PiEstimation only had one stage, there is only single data points in each figure. As can be seen, RS was significantly reduced while decreasing std for task number.

Note that, as PiEstimation application used a large number as partition number in default setting, enabling speculative execution increased the scheduling cost for the application due to a large number of tiny tasks (task execution time was very short), and resulted in 22% performance reduction compared to default setting.

C. Group III: Applications with no imbalance problem

K-Means and LogisticRegression belonged to this group. Both of them had been predicted to have no imbalance problem. For K-Means, Figure 3h and Figure 4h show RS and std for task number in each stage of the sample application and whole application. From the aspect of RS, the prediction error was found to be small. From the aspect of std for task number, the prediction was not accurate.

For LogisticRegression, Figure 3i and Figure 4i show RS and std for task number in each stage of the sample application

(a) TC (b) PR (c) CC

(d) WC (e) TRC (f) SSSP

(g) PI (h) KM (i) LR

Fig. 3: Ratio of Task Straggler (RS)

and whole application. From the aspect of RS, this prediction was not accurate, although the actual imbalance problem was not significant. From the aspect of std for task number, the prediction was correct.

V. Conclusion

This paper presents a model driven approach for predicting and addressing possible task straggler and skewed task distribution problem. Evaluation on two different clusters demonstrates that the model can correctly predict such problems with high accuracy, allowing the algorithm to suggest near-optimal partition number and locality settings. Experimental result demonstrates the effectiveness of our approach in improving performance by up to 71% on Cluster I and up to 68% on Cluster II. We believe that our approach will allow users to tune their systems while revealing the root cause behind the suboptimal performance problems.

Acknowledgment

This material is based upon work supported by the Air Force Office of Scientific Research award number FA9550-15-1-0164 under the DDDAS program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agency.

References

[1] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin *et al.*, "Apache spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[3] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 293–307.

[4] P. Garraghan, X. Ouyang, R. Yang, D. McKee, and J. Xu, "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters," *IEEE Transactions on Services Computing*, 2016.

[5] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative api for real-time applications in apache spark," in *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018, pp. 601–613.

[6] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 69–84.

[7] S. Li, M. T. Amin, R. Ganti, M. Srivatsa, S. Hu, Y. Zhao, and T. Abdelzaher, "Stark: Optimizing in-memory computing for dynamic dataset collections," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 103–114.

[8] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *Computer Communications, IEEE*

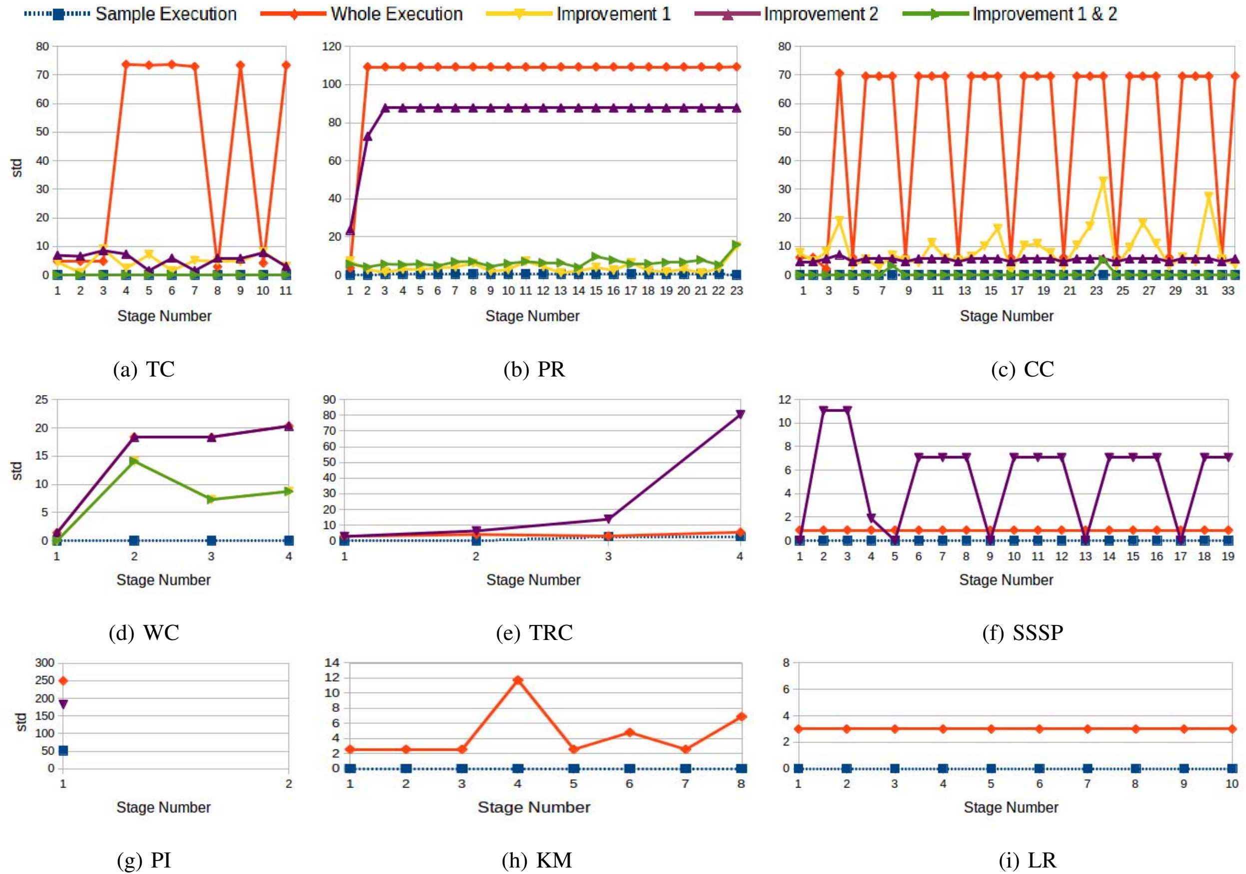


Fig. 4: Standard Deviation (std) for Task Number

INFOCOM 2016-The 35th Annual IEEE International Conference on. IEEE, 2016, pp. 1–9.

[9] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: a constraint-aware scheduler for heterogeneous datacenters," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 977–987.

[10] K. Ousterhout, C. Canel, M. Wolffe, S. Ratnasamy, and S. Shenker, "Performance clarity as a first-class design principle," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. ACM, 2017, pp. 1–6.

[11] K. Wang, M. M. H. Khan, N. Nguyen, and S. Gokhale, "Design and implementation of an analytical framework for interference aware job scheduling on apache spark platform," *Cluster Computing*, pp. 1–15, 2017.

[12] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[13] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica, "Effective straggler mitigation: Attack of the clones." in *NSDI*, vol. 13, 2013, pp. 185–198.

[14] Q. Chen, J. Yao, and Z. Xiao, "Libra: Lightweight data skew mitigation in mapreduce," *IEEE Transactions on parallel and distributed systems*, vol. 26, no. 9, pp. 2520–2533, 2015.

[15] Z. Liu, Q. Zhang, R. Boutaba, Y. Liu, and B. Wang, "Optima: on-line partitioning skew mitigation for mapreduce with resource adjustment," *Journal of Network and Systems Management*, vol. 24, no. 4, pp. 859–883, 2016.

[16] W. Wang, K. Zhu, L. Ying, J. Tan, and L. Zhang, "Maptask scheduling in mapreduce with data locality: Throughput and heavy-traffic optimality," *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 190–203, 2016.

[17] X. Ma, X. Fan, J. Liu, H. Jiang, and K. Peng, "vlocality: Revisiting data locality for mapreduce in virtualized clouds," *IEEE Network*, vol. 31, no. 1, pp. 28–35, 2017.

[18] T. Chiba and T. Onodera, "Workload characterization and optimization of tpc-h queries on apache spark," in *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 112–121.

[19] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 153–167.

[20] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *NSDI*, vol. 2, 2017, pp. 4–2.

[21] Y. Le, J. Liu, F. Ergun, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 2004–2012.

[22] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[23] Apache hadoop. [Online]. Available: http://hadoop.apache.org/

[24] A. Vitorovic, M. Elseidy, and C. Koch, "Load balancing and skew resilience for parallel joins," in *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*. IEEE, 2016, pp. 313–324.

[25] T.-D. Phan, S. Ibrahim, A. C. Zhou, G. Aupy, and G. Antoniu, "Energy-driven straggler mitigation in mapreduce," in *European Conference on Parallel Processing*. Springer, 2017, pp. 385–398.

[26] Stanford snap. [Online]. Available: http://snap.stanford.edu/

[27] Sloan digital sky survey. [Online]. Available: http://www.sdss.org/