Hierarchical Spark: A Multi-cluster Big Data Computing Framework

Zixia Liu, Hong Zhang, and Liqiang Wang
Department of Computer Science
University of Central Florida, Orlando, FL, USA
{*zixia, hzhang1982*}*@knights.ucf.edu, lwang@cs.ucf.edu*

***Abstract*—Nowadays, with the increasing burst of newly generated data everyday, as well as the vast expanding needs for corresponding data analyses, grand challenges have been brought to big data computing platforms. Computing resources in a single cluster are often not able to fulfill the computing capability needs. The requests of distributed computing resources are dramatically arising. In addition, with increasing popularity of cloud computing platforms, many organizations with data security concerns are more favor to hybrid cloud, a multi-cluster environment composed by both public cloud and private cloud in purpose of keeping sensitive data local. All these scenarios show great necessity of migrating big data computing to multi-cluster environment.**

In this paper, we present a hierarchical multi-cluster big data computing framework built upon Apache Spark. Our framework supports combination of heterogeneous Spark computing clusters. With an integrated controller within the framework, it also facilitates ability for submitting, monitoring, executing of Spark workflow. Our experimental results show that the proposed framework not only enables possibility of distributing Spark workflow throughout multiple clusters, but also provides significant performance improvement compared to single cluster environment by optimizing utilization of multi-cluster computing resources.

Key Words: Hybrid Cloud, Hierarchical, Multi-cluster, Big data, Spark

I. INTRODUCTION

With the coming and on-going duration of the information era, more and more data are generated everyday, even in an exploding speed. These data carry lots of invaluable information that are of great importance to human society and global development. The necessity of analyzing such drastic amount of big data stimulates the continuing prosperous development of big data computing. Since the analytical process of such data are way over the computational capability of even the best single computing node, people spend great efforts to develop parallel computing methods and platforms. Among them, Apache Hadoop and Apache Spark are two of the most popular open-source big data computing platforms. Apache Hadoop utilizes HDFS (Hadoop Distributed File System) as its storage layer, and uses MapReduce computing model to provide end users a distributing computing platform that has better reliability and scalability than traditional parallel computing interfaces. Apache Spark further improves the performance of Apache Hadoop by introducing RDD (Resilient Distributed Dataset) object based on the in-memory technique. Apache spark overall provides better distributed computing performance for big data analytical workflow, which supports much richer computational operations and more complicated workflow structure comparing to Apache Hadoop.

Both Apache Hadoop and Apache Spark are deployed upon the concept of cluster. All computing nodes form a cluster that can be employed by the resource manager such as YARN to schedule computing tasks. In this case, all internal nodes are natively considered to have a local network connection with other nodes in the cluster. However, distributed computing may involve multiple geographical locations. Even if we use the virtual private network (VPN) technique to connect these computer into a single cluster, as the resource manager is not able to detect and realize such heterogeneous network structure, the cluster performance will degrade significantly. A hybrid cloud is a good example to utilize distributed computing on multiple geographical locations. A user may have one local cluster initially, but then realizes the shortage of computing resources due to data analytical demand, and decides to request more resources from public cloud platforms such as Amazon EC2. The user is then facing with the obstacle of how to integrate and utilize the resources from both local private cluster and public cloud computing recourses. On the other aspect, with ongoing popularity of cloud computing platforms, many organizations with data security concerns would like to keep sensitive data local. In this scenario, it is of great significance to enforce the isolation between multiple clusters, in order to obey the data security standard. For example, the data security standard may grant only the transferring of computation generated intermediate data but not original input data. These scenarios motivate great necessity of migrating big data computing workflow to multi-cluster environment.

In this paper, we present a multi-cluster big data computing framework built upon Spark. Our major contributions include:

- A framework that addresses the problem of utilizing the computation capability provided by multiple Apache Spark clusters, where heterogeneous clusters are also permitted.
- A scheduling algorithm to optimize workflow execution on our multi-cluster big data computing framework.
- An integrated controller within the framework, which grants ability for submitting, monitoring, and finishing of workflows.

II. RELATED WORK

MapReduce is a computational model with great popularity. Apache Hadoop is a popular open-source implementation of the MapReduce paradigm. There are several existing research projects either providing a hierarchical level design for the MapReduce model or even providing framework design support for deploying MapReduce jobs to multiple cluster

environments. [17] proposes a new Map-reduce-Merge model as an extension of the MapReduce paradigm. The new merge phase can merge heterogeneous dataset already partitioned and sorted by MapReduce and can express relational algebra operators as well as join algorithms. However, it increases system complexity and learning curve due to the introduction of several new components. [11] classifies MapReduce jobs into two categories based on whether they are recursively reducible or not. It provides a solution that could support hierarchical reduction or incremental reduction for recursively reducible jobs, however it is only applicable to single cluster environment. [12] introduces MRPGA (MapReduce for Parallel Genetic Algorithms), which additionally adds a second reduce phase to the original MapReduce model in order to address genetic algorithms, however, this extension is designed for a special application and may not be suitable as a solution for general MapReduce applications. The concept of "distributed MapReduce" is introduced in [14], which is a hierarchical design for MapReduce. However this solution is lacking scheduling algorithm and programming model design. [16] addresses the data analysis problem in the hybrid cluster, which consists of a local cluster and cloud computing resources, with the usage of both local and global reduce phases. However, the solution also lacks scheduling algorithm. Luo *et al.* [1],[9] presents a hierarchical MapReduce framework that adopts the Map-Reduce-GlobalReduce model introduced in the paper. The framework is capable of utilizing computational resources from multiple clusters to collaboratively accomplish MapReduce jobs, and it also provides scheduling algorithms for compute-intensive jobs and data-intensive jobs.

However, all above solutions are targeted for MapReduce paradigms, and are not designed for Apache Spark system. To efficiently execute an application on hierarchical Spark framework, deciding or estimating the computational workload for each component job is a significant measurement for scheduling algorithms. In Hadoop, the component job workload is usually propotional to its assigned input size. However for Spark, since the distributed computation may be entirely different, this assumption no longer holds. In this case, a performance model is needed to provide computing load estimation based on the jobs and cluster environment. To the best of our knowledge, research work related to Spark performance model is still relatively lacking. We found in total four references [4], [5], [6], [8], which discuss topics related to Spark performance model, it is certainly a research direction with future potential. In this paper, we propose a performance model to be used in our experiment.

III. Architecture of Hierarchical Spark

The architecture of hierarchical Spark mainly contains two component layers, the global controller layer and the distributed layer. The global controller layer consists of the workflow scheduler and the global listener. The distributed layer consists of the distributed daemons, each has job manager and job monitor in charge of submitting and monitoring the job allocated to corresponding cluster.

When using hierarchical Spark, users provide the following files to the framework: (1) application files that contains component jobs to form the overall workflow; (2) configuration files that specifies application dependencies; (3) profiling information for component jobs. If the profiling information is absent, it can be supplemented by on-site profiling with the original workflow and sample data. Upon receiving input, the workflow scheduler extracts information, and then generates job allocation arrangement by our scheduling algorithm. Once the arrangement is decided, the workflow scheduler will utilize the global listener to distribute and start the actual execution. The global listener is another component of the global controller layer, based on the scheduling plan provided by the workflow scheduler, the global listener will deploy corresponding job to assigned clusters. It will also communicate with distributed daemons, which manage job submission and monitor job status. Once job finished, the distributed daemon will notify the global listener so that the latter will arrange transferring of the output file, submitting other dependent jobs, or launching the final job, until the entire workflow is finished. The architecture of hierarchical Spark is illustrated in Figure 1.

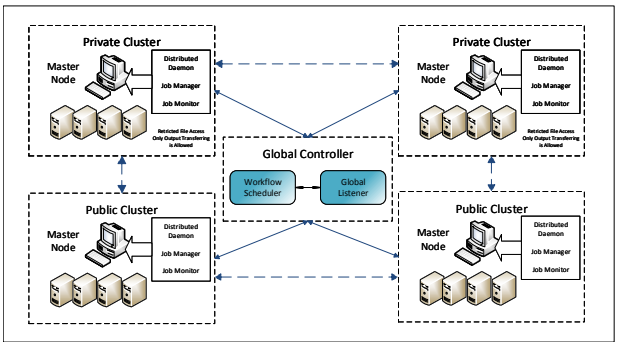

Fig. 1. Architecture of Hierarchical Spark

A. Workflow Model

In hierarchical Spark, the workflow model contains three type of components, *i.e.,* non-dependent job, dependent job, and final job. The non-dependent jobs are those jobs that start from initial input files, and have no other dependencies. The dependent jobs are those jobs that have dependencies on other jobs, either non-dependent or dependent ones. The final job is the last component in the entire workflow, this is fixed to be executed on the central cluster. By dependencies, all these component jobs form the entire workflow as the input of our framework. Each job is a basic element that will be scheduled to clusters for computation. Figure 2 illustrates a hierarchical Spark workflow.

Recall in Spark, a job can be expressed as a DAG (Directed Acyclic Graph), similarly, our framework workflow can also be represented as a DAG. The transformation from original spark workflow to the our framework workflow is natural.

Basically, Algorithm 1 starts by letting each stage in initial spark DAG become a job in the framework workflow. Then,

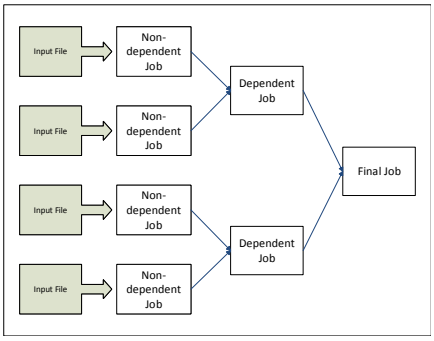

Fig. 2. Illustration of Hierarchical Spark Workflow

we suggest splitting jobs in the workflow into multiple ones, or combine several jobs in the framework workflow into one job. We recursively examine the workflow until there is no new change. For example, in lines 3-8 of Algorithm 1, for a job in the workflow, if its "inputsize" over "blocksize" (equivalently the number of blocks) is very big, we suggest split it into multiple jobs, with default suggesting splitting number shown in algorithm. Conversely, in lines 11-18, by looking at some job i together with its dependency jobs list D_i as a group, we can check whether using the whole group as one job in our workflow is beneficial, then apply the change if this will significantly lower total transit data meanwhile not violating other constraints. We provide general default values to thresholds in the algorithm, for example, the default value of threshold1 is 10×(max # of executors in all clusters). However they can also be specified by the user to customize the suggestion engine.

Using the wordcount application as an example. When it is used as the workflow input for our framework, Algorithm 1 will provide transformation plan from this Spark workflow to our framework workflow. For this example, if the number of block for the input file is larger than threshold1, we suggest split the job into multiple ones. In this case, the split can be accomplished by roughly repeating the unary operation "reducebykey()" twice, with each new non-dependent job taking care of one portion of the initial input file.

The core pesudo code for original wordcount application is follows.

$$line.split(\text{“ ”}).map(word \rightarrow (word, 1)).reducebykey()$$

Core pesudo code for our framework wordcount application (non-dependent jobs) is the same with the code above, which also demonstrates that the transformation burden is little to framework users. The core pseudo code for the final job is:

$$collectedresultline.map(parser(\text{“}K, V\text{”} \rightarrow (K, V))) .reducebykey()$$

Algorithm 1 Spark Workflow Transformation Algorithm

1: Let each stage in Spark become a job
2: **for** each job i **do**
3: **if** ($D_i = \emptyset$ && (β =# of $cluster_{hasinput}$)>1 && input/blocksize > threshold1) **then**
4: split job to β jobs, update corresponding dependency lists
5: **end if**
6: **if** ($D_i \neq \emptyset$ && input/blocksize > threshold1) **then**
7: split job to input/(blocksize×threshold1) jobs, update corresponding dependency lists
8: **end if**
9: **end for**
10: Change=true
11: **while** Change==true **do**
12: Change=false
13: **for** each job i **do**
14: **if** *!IsNewSplittedJob*(i) && total input blocks to i <threshold1 && new output/original total output<threshold2 && combined input exists if non-dependent jobs are involved **then**
15: combine job i and D_i into one job, update corresponding dependency lists; Change=true;
16: **end if**
17: **end for**
18: **end while**

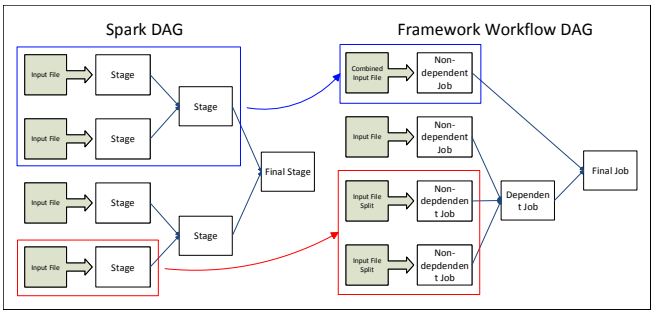

Fig. 3. Illustrative Framework Workflow DAG Generation

This final job is added to act as an eventual collection and reduce procedure for all intermediate data generated by previous non-dependent jobs, its code is straightforward and easy for framework users to add.

Figure 3 illustrates a more general case, by showing the original spark DAG on the left and Framework workflow DAG on the right. It shows that the general suggestion result from our algorithm which may include some splitting as well as combing, with other stages in the original DAG directly becoming corresponding jobs in our new workflow.

IV. Scheduling Algorithm

Our framework not only aims at enabling distributing component jobs of an entire workflow to multiple spark clusters for cooperated computing, but is also equipped with scheduling algorithm designed to better achieve multi-job & multi-cluster

scheduling in purpose for better performance. Our proposed algorithm is shown in Algorithm 2.

Multi-job & multi-cluster job scheduling is a well-known NP-hard problem. To achieve a good solution in an efficient way, we use simulated annealing as the major heuristic algorithm for solution searching. To further increase the efficiency, we use greedy algorithm to achieve a better initial solution that will be provided as input to the simulated annealing algorithm.

Regardless of the choice of heuristic algorithms, the core design of our scheduling algorithm is nonetheless the evaluation function that could assess the scheduling arrangement. Since our aim is to reduce the total execution time of the entire workflow, our evaluation function is designed to be capable of evaluating the running time cost of a specific scheduling arrangement. Further, the evaluation function needs a performance model, which can provide us an estimation for the running time of a job on a cluster.

A. Performance Model

Now, we provide the performance model that could estimate the running time of a job on a candidate cluster. To better introduce the entire performance function, we first introduce the performance model for a stage in a job. Its details are as follows:

Let l be the average computing time of a task in a stage using its required executor. We define it in unit of second. Let c denote the total available executors in a cluster for this job. Let a denote the number of tasks in a RDD in a stage, we choose the maximum number of tasks in a RDD in a stage if RDDs in a stage have different number of tasks.

Thus, a/c represents possible waves during execution.

The performance model for a stage is defined as:

$$t_{stage} = l \cdot a/c \tag{1}$$

Now, we propose the performance function that models the execution of a job into a more detailed level as running of stages, relevant to the stage running concept in Spark.

$$PF = \frac{InputSize}{SampleSize} \times (\sum_{stages} t_{stage} + \sum_{shuffles} ST * nF) \tag{2}$$

where ST is the intermediate data shuffle time that happens between stages. If no shuffle exists between some stages, corresponding shuffle time equals zero. nF is the network factor, which can reflect the different internal network speed of cluster. Notice that we consider the possibility that the profiling may be corresponding to a sampling data, instead of the entire input data, so the ratio of $InputSize$ over $SampleSize$ is also considered in the formula.

This performance function is currently adopted in our framework. Nonetheless, we would like to point out that, when applicable, depending on different coarseness of profiling data, the performance function can certainly be replaced by even more specially designed or more complicated performance models suitable for certain scenarios corresponding to the actual application.

B. Scheduling Algorithm and Evaluation Function

When designing the scheduling algorithm for our framework, we have considered different options at the early stage. The non-dependent jobs in our framework are a little special, as each of them has no dependencies, thus all can be submitted at the beginning of the execution. Consider these jobs as a group, one option for scheduling is the initial optimizing scheme designed to focus on optimizing the arrangements of this group for better performance. As stated before, to decide an optimized plan for all jobs in this group, we use greedy algorithm (sort all stages with computing load in descending order) to achieve an arrangement plan, and then an on-the-fly arrangement plan for other jobs in the workflow in actual submission time order. In this scheme, the arrangement plan for all non-dependent jobs will not consider the consequences it brought to other dependent jobs which depends on them, it therefore can be seen as a non-forward-looking scheme.

Stimulated from this idea, but further improved, we design a global optimizing scheme aims at providing an overall scheduling arrangement of the entire workflow. We want to take into consideration the consequential effect of optimizing non-dependent jobs it may bring to the later jobs. In other words, instead of providing current moment optimized arrangement plan, we would like to take global vision, foresee the whole picture of workflow DAG, then decide an arrangement plan for each job in the workflow. Due to its more advanced feature, we select this scheme as our framework scheduling scheme. The entire arrangement plan will be decided based on the following procedure:

Firstly, similar as proposed in the initial optimizing scheme, we obtain arrangement plan for all non-dependent jobs by greedy algorithm. Now, for each non-dependent job, the scheduling plan for it will be represented as $(cluster, t_{start}, t_{finish})$, where $cluster$ is the selected cluster, t_{start} is the job start time. t_{finish} is the job finish time generated by the performance function.

For all dependent jobs, the tuple $(cluster, t_{start}, t_{finish})$ can be filled in by first calculating job_{st} using equation 3:

$$job_{st}[s'][j] = \max(t', cluster_{at}[j]) \tag{3}$$

where $t' = \max_{s}\{t_{finish} \ of \ s + IMO(s, s') | s \in dep \ set \ of \ s'\}$, $IMO()$ is the intermediate output transferring time for two jobs, $cluster_{at}[j]$ is the available time of cluster j which s' may depoly to. Equation 3 is also used for non-dependent jobs by considering their dependency set as empty and only use clusters that has its input.

For any job where its dependencies has been cleared, its $job_{st}[s'][]$ times are available. All these jobs can gradually be fit into the greedy algorithm. Once their cluster arrangements are decided, the corresponding t_{start}, and t_{finish} components can be filled in. Recursively, this can form an entire initial scheduling plan for the entire workflow, this part is shown in lines 3-33 in our proposed Algorithm 2. In fact, during this process, we are already simulating the execution process of the workflow together with the usage of greedy algorithms

for achieving an initial scheduling plan. The similar idea of simulation will be used as the evaluation function in the iteration process of the simulated annealing algorithm, where t_{finish} of the final job is the eventual output of the evaluation function.

Secondly, for simulated annealing, each time it will randomly change one cluster arrangement if suitable, feed into the evaluation function $E()$ to simulate its execution to achieve the new t_{finish} of the final job, which is the result of the evaluation function $E()$. Based on its result and therefore the result of the function $P()$, the simulated annealing can decide whether to keep the current plan or to update to the new one. The idea of the simulated annealing is that, if the new arrangement is better than the current one, it will always accepts it; however, if the new arrangement is worse than the current one, it will stochastically accepts it, which helps in jumping out of the "local" extrema. The probability depends on the parameter αT and the evaluation difference. At the beginning of simulated annealing, the result of $exp()$ function if used is very close to 1 and therefore there is much higher chance for accepting bad arrangements. In contrary, when nearing to the end of the algorithm, T, therefore αT, becomes small and the chance of accepting bad arrangement is significantly decreased. This simulated the physical annealing process where T act as the "temperature" in original process. This algorithm is good at obtaining global extreme for scheduling arrangement. Parameter α is used to make sure that according to the range of evaluation function difference, the initial probability to accept bad arrangement is very close to 1. The function $P()$ used in our simulatedAnnealing() function is:

$$P(E(S), E(S_{new}), T) = \begin{cases} 1 & IF\ E(s_{new}) \leq E(s) \\ \exp\left(\frac{E(s)-E(s_{new})}{\alpha T}\right) & Otherwise \end{cases} \tag{4}$$

The SimulatedAnnealing() function is shown in lines 35-43 in Algorithm 2. To supplement the detail, we now will state the definition of the evaluation function $E()$, which is a simulation process, as follows:

The input for the simulation engine is the set of jobs, each with a configuration tuple $(cluster, t_{start})$. The simulation process will then generate a simulated running for each cluster. For each cluster, the input is the tuples with format (job, t_{start}). The simulated cluster submit the job to it by order of start time. For all jobs with t_{start} time undecided, the cluster simulation will wait until its dependencies are all cleared. Then each job on the cluster simulation will simulate its running by result from the performance function, with currently updated cluster information being considered. Simulations for all clusters are processed simultaneously. Eventually, all simulation of all clusters are finished, then the t_{finish} of the final job becomes the eventual output of the evaluation function. An illustration graph of this process can be found in Figure 4. Now, we can formally state our scheduling algorithm in Algorithm 2.

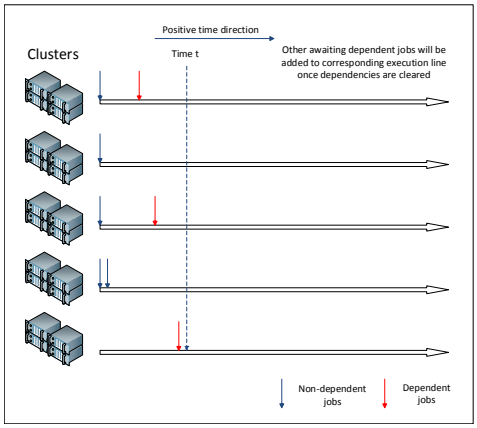

Fig. 4. Cluster Running Simulation

V. Implementation Issues

A. Global Controller and Distributed Daemon

To enable the distributing, monitoring and executing of the big data analytical workflow, we designed global controller and distributed daemon in our proposed multi-cluster big data computing framework.

The global controller is composed by the workflow scheduler and global listener, the workflow scheduler accepts user provided framework input, extract necessary information, and provide the scheduling arrangement result. The global listener is constructed as a multi-threading program, where all distributed daemon will be connected to it. Functionality of the global listener include: Send job arrangement to corresponding distributed daemon, order the moving of intermediate files.

The distributed daemon are composed by the job manager and the job monitor. Upon receiving job arrangement, the distributed daemon will submit the job to the cluster, it will also capture the application ID after submission, using it to monitor the job status. Once succeed, the distributed daemon will notify the global listener about the status update and intermediate file collection movements will be applied if necessary.

B. File Transfer

During the execution of the workflow, there are certain steps that contain or require intermediate data file transfer. For example, the final job definitely relies on outputs generated by some other jobs in the workflow. Also, when deploying dependent jobs, it may be necessary to apply movement of intermediate data to other clusters. In our framework, we assume the underlying file system to be HDFS (Hadoop Distributed File System). In order to make the file transfer more efficient, instead of applying the HDFS to local, transfer, then local to HDFS procedure, we use transfer command provided by the HDFS API (hdfs distcp) to facilitate direct and efficient parallel file transfer between source and target HDFS system.

Algorithm 2 Scheduling Algorithm

Create *Scheduling*[], cluster available time $cluster_{at}$[]. For each job i, create estimated job start time $job_{st}[i][]$, dependency list $D[i][]$, and the wall clock time for all jobs in dependency lists of job i, *i.e.*, *Dep-walltime*$[i][]$. For each cluster j, create *Running*$[j][]$ to record running interval and capability usage of each job on cluster

GreedySolution() {
while RemainingJob!=0 **do**
  for each job i where *Scheduling*$[i]$=Null **do**
    if $D[i][]$==Null **then**
      scheduling-pool.add(job i)
      $job_{st}[i][]$= Eqn(3) result by *Dep-walltime*$[i][]$
    end if
  end for
  if All jobs in pool has same min start time **then**
    Sort(scheduling-pool, computing load, descending)
  else
    Sort(scheduling-pool, $\min_j(job_{st}[i][j])$, descending)
  end if
  for each job i in scheduling-pool **do**
    initialize *Score*[];
    for each candidate cluster j **do**
      // Using reciprocal of estimated job finishing time as score for cluster j w.r.t job i for scheduling
      decide cluster j capability by *Running*$[j][]$ and $job_{st}[i][j]$
      Score$[j]$=$1/(job_{st}[i][j] + PF(i,j))$
    end for
    j=*Scheduling*$[i]$=$argmax$(*Score*[])
    RemainingJob$--$
    $job_{walltime} = job_{st}[i][j] + PF(i,j)$
    update *Running*$[j][]$
    if cluster j is fully occupied by submitting job i **then**
      $cluster_{at}[j] = \min(job_{walltime})$ for current jobs on j
    end if
    Delete job i from all dependency lists, adding $job_{walltime}$ to corresponding *Dep-walltime*[][]
  end for
end while
return Scheduling }

SimulatedAnnealing() {
for k = 0 through kmax **do**
  $T = 100 \times (\sqrt[kmax]{0.001})^k$
  Scheduling' = randomAlternation(Scheduling)
  if P(E(Scheduling), E(Scheduling'), T) >= rand(0, 1)) **then**
    Scheduling=Scheduling'
  end if
end for
return Scheduling }

VI. Experiments

Our experiments are done on Amazon EC2 cloud computing platform, the Hadoop version is 2.7.3, and the Spark version is 2.1.0. Each cluster used in the experiment utilizes at most 9 m4.xlarge computing nodes to compose cluster of different sizes. For each cluster we mention below, the number of nodes are referring to data nodes (computing nodes) in the cluster and there will be an extra name node in the cluster as well. We set one executor on each computing node that utilizes four virtual cores. There are mainly two purposes of our experiments. First, to show that by enabling multi-cluster collaborative execution, we could dispatch the original workflow by component jobs that could run on different clusters. Second, in some situations, the distributed workflow can also outperform the original application due to enabling of computing resources from multiple clusters and our designed scheduling algorithm. Our first experiment uses the WordCount application.

- WordCount: Comparing Effort of Original and Distributed workflow

In the first experiment, we run WordCount on 100GB input file with a 6-node cluster, this will act as our execution for the original workflow and as the comparing case for other distributed workflows.

For comparison, we run the distributed workflow on two, three, and four clusters (one of them is the central cluster where the final job is on), each having 6 computing nodes, and each deals with its proportional portion of the original total input. We further assume all clusters have all inputs in this experiment. The distributed component jobs are the same as the original WordCount job, however, in the end, the output files will be collected to the main 6 nodes cluster, and apply an additional application which act as a global reduce process. For 100 GB input on original workflow on one single cluster with 6 computing nodes, the total execution time is 16 minutes. As an example, in comparison, for three cluster scenarios, the 33 GB input on 6-node cluster costs a maximum of 5.6 min execution time, the generated output file is around 2 MB for each cluster, gathering them to the main cluster using HDFS distcp will cost around 20s, and the final job running time on the 6-node main cluster will cost about 49 seconds. Due to the scale of the workflow, other overhead caused by the architecture is low enough to be ignored, in fact, the scheduling algorithm can even be omitted in this special case. Therefore, the total performance comparison is 16 minutes vs 6.8 minutes, which yields a 2.35 times speed up. The result related to all number of clusters is shown in Figure 5.

- WordCount: Comparing Effect of Default and Our Proposed Scheduling Algorithm

In the second experiment case for WordCount, the scenario simulates where there are four clusters, with 2, 4, 6, 8 computing nodes respectively. The 8 nodes cluster is the central cluster where the final job is on. Four identical component jobs are split from the original WordCount computing, each deal with 1/10, 1/5, 3/10, 2/5 portion of the 100G input, proportional to the component cluster's computing node numbers. The entire

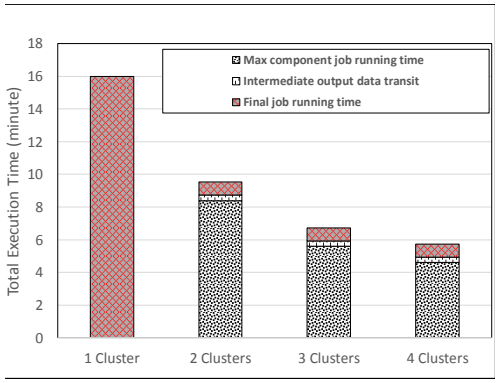

Fig. 5. Wordcount Workflow Execution Time Comparison

computation process is the same as in the first experiment. We further assume all clusters have all inputs in this experiment. Now, suppose we adopt the default fair scheduling algorithm in Spark to our framework. If all workloads are in descending order of their input size (as well as computing burden in this case), but all cluster are in ascending order of their number of nodes, then by fair scheduling, the heaviest task will be arranged to the smallest cluster, etc. However, for our scheduling algorithm, no matter what the sequence of the workloads and clusters are, the scheduling will make the correct decision to send corresponding component jobs to the cluster that is proportional to its computing load. We now show the experiment result in Table I.

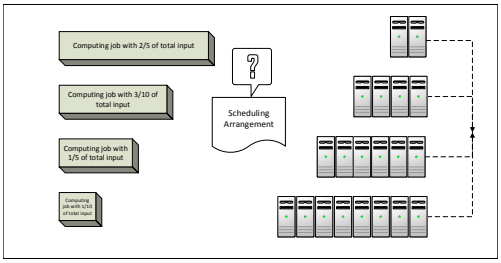

Fig. 6. Illustration Graph for Component Jobs and Clusters

TABLE I
COMPONENT FINISHING TIME WITH DIFFERENT SCHEDULING SCHEMES

	Fair Scheduling		Proposed Scheduling	
Cluster	Input	Finish Time	Input	Finish Time
cluster-1 (2 nodes)	2/5	19 mins	1/10	5.0 mins
cluster-2 (4 nodes)	3/10	5.2 mins	1/5	5.2 mins
cluster-3 (6 nodes)	1/5	5.2 mins	3/10	5.2 mins
cluster-4 (8 nodes)	1/10	5.3 mins	2/5	5.3 mins

We can observe from the result that, for the default fair scheduling scheme, the longest component job running time is 19 minutes, whereas for our proposed scheduling algorithm, the longest component time is 5.3 minutes, since all intermediate outputs from all components jobs are all very similar in sizes (about 2 MB), and the running time for the final process job will be very similar as well, the running time improvement in the component jobs will greatly be reflected in the overall execution time. Therefore, our proposed scheduling algorithm is better than the default scheduling scheme in Spark. In fact, for component jobs only, the maximum running time achieves a 3.58 times speedup by scheduling arrangement improvement.

- GIS Analytical Workflow: A Practical Workflow Demonstration on Hierarchical Spark

In [2] and [3], some GIS (Geographical Information System) computations have been accomplished on parallel computing platforms, especially in [3], these computations are executed on Apache Spark platform. Such computations include geographic mean computation, geographic median computation, etc. Stimulated by such application, in this experiment, we try to distribute an actual GIS workflow to multiple clusters by our framework. The workflow uses users' twitter sending GPS positions with format (userID, lon, lat) as input, accumulated by user ID, calculate their geographic mean and median, then join the results by user ID again to achieve a tuple for each user that could describe the geographical social behavior center for the user for further analysis, the output format is (userID, Geographic Mean, Geographic Median). The definition of geographic mean and median, together with a workflow illustration graph in Figure 7 is shown below:

Geographic Mean:

$$LON = \frac{\sum_{i=1}^{n} lon_i}{n} \qquad LAT = \frac{\sum_{i=1}^{n} lat_i}{n} \tag{5}$$

Geographic Median:

$$Median = \min_{x \in space} \sum_{i=1}^{n} \sqrt{(x_{lat} - lat_i)^2 + (x_{lon} - lon_i)^2} \tag{6}$$

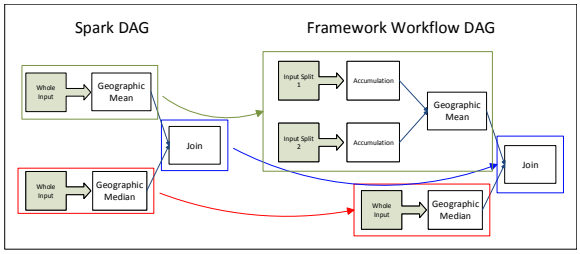

Fig. 7. Illustration Graph for Spark DAG Workflow and Framework DAG Workflow

This experiment shows a scenario with three component clusters. The first cluster has 4 nodes and is a private cluster with half of the whole input data, which are sensitive. The second cluster has 4 nodes and is a public cluster with second half of non-sensitive data. The third cluster has 8 nodes (central cluster) and is a private cluster with whole input. In the framework workflow, since the geographic median computation requires whole input, it is kept as one job and deployed to third cluster. The original geographic mean computation is firstly split into two accumulation component jobs, with each one takes half of whole input on the first and

second cluster respectively, accumulates their GPS locations and number of occurrence, generate output in format (userID, accumulated lon, accumulated lat, number of occurrence). Then, the geographic mean job in our framework workflow which reduces the intermediate outputs from two accumulation jobs is launched on the first cluster. Eventually, both intermediate outputs from geographic mean and geographic median jobs are gathered to central cluster, where the joining of the two intermediate results by userID is launched to achieve final result desired.

The execution time comparison is shown in Table II. It shows that our framework not only enables the collaborative execution of this workflow on multiple clusters, but also achieves performance improvement comparing to the original single cluster execution. It is also worth mentioning that this experiment well demonstrates the capability of our framework in maintaining certain data security and isolation standards.

TABLE II
COMPONENT JOB FINISHING TIME AND TOTAL EXECUTION TIME OF FRAMEWORK WORKFLOW (IN COMPARISON, TOTAL EXECUTION TIME IN ONE CLUSTER IS 3.3 MIN)

Framework workflow	Time
Accumulation 1	55 s
Accumulation 2	59 s
Geographic Mean	23 s
Geographic Median	1.5 min
Final Join	45 s
Total Execution	2.8 min

VII. CONCLUSIONS AND FUTURE WORK

We design a scheduling algorithm basing on the heuristic simulated annealing approach. The experiments shows that, our framework not only enables the functionality to distribute original spark workflow to multiple clusters for collaborative execution, it also provides great performance improvement due to better utilization of the overall computing resources.

In the future work, we would like to focus on extending the functionality of the framework that could support iterative computation workflow, this shall be reflected in a more complicated global controller that could coordinate inter-framework-calling file transfer and could iteratively call framework for each iteration.

VIII. ACKNOWLEDGEMENT

This work was supported in part by NSF-CAREER-1622292.

REFERENCES

[1] Y. Luo, B. Plale, Z. Guo, W. Li, J. Qiu and Y. Sun, "Hierarchical MapReduce: towards simplified crossdomain data processing." Concurrency and Computation: Practice and Experience 26.4 (2014): 878-893.

[2] Z. Hong, Z. Sun, Z. Liu, C. Xu, and L. Wang, "Dart: A geographic information system on hadoop." 2015 IEEE Cloud Computing (CLOUD).

[3] Z. Sun, Z. Hong, Z. Liu, C. Xu, and L. Wang, "Migrating GIS Big Data Computing from Hadoop to Spark: An Exemplary Study Using Twitter." 2016 IEEE Cloud Computing (CLOUD).

[4] K. Wang, and M.M.H. Khan, "Performance prediction for apache spark platform." 2015 IEEE HPCC.

[5] G.P. Gibilisco, M. Li, L. Zhang, D. Ardagna, "Stage aware performance modeling of DAG based in memory analytic platforms." 2016 IEEE Cloud Computing (CLOUD).

[6] K. Wang, M.M.H. Khan, N. Nguyen, and S. Gokhale, "Modeling Interference for Apache Spark Jobs." 2016 IEEE Cloud Computing (CLOUD).

[7] V. Subramanian, L. Wang, E.J. Lee, and P. Chen, "Rapid Processing of Synthetic Seismograms Using Windows Azure Cloud", IEEE CloudCom, 2010.

[8] G. Wang, J. Xu, and B. He. "A Novel Method for Tuning Configuration Parameters of Spark Based on Machine Learning." 2016 IEEE HPCC.

[9] Y. Luo, Z. Guo, Y. Sun, B. Plale, J. Qiu, and W.W. Li, "A hierarchical framework for cross-domain MapReduce execution." Proceedings of the second international workshop on Emerging computational methods for the life sciences. ACM, 2011.

[10] V. Subramanian, H. Ma, L. Wang, E.J. Lee, and P. Chen, "Rapid 3D Seismic Source Inversion Using Windows Azure and Amazon EC2", IEEE SERVICES, 2011.

[11] M. Elteir, H. Lin, and W. Feng, "Enhancing mapreduce via asynchronous data processing." , 2010 IEEE Parallel and Distributed Systems (ICPADS).

[12] C. Jin, C. Vecchiola, and R. Buyya, "MRPGA: an extension of MapReduce for parallelizing genetic algorithms." eScience, 2008. eScience'08. IEEE Fourth International Conference on. IEEE, 2008.

[13] H. Huang, L. Wang, B.C. Tak, L. Wang, and C. Tang, "CAP3: A Cloud Auto-Provisioning Framework for Parallel Processing Using On-Demand and Spot Instances", 2013 IEEE Cloud Computing (CLOUD).

[14] M. Cardosa, C. Wang, A. Nangia, A. Chandra, and J. Weissman, "Exploring mapreduce efficiency with highly-distributed data." Proceedings of the second international workshop on MapReduce and its applications. ACM, 2011.

[15] H. Zhang, H. Huang, and L. Wang, "MRapid: An Efficient Short Job Optimizer on Hadoop", IEEE IPDPS 2017.

[16] T. Bicer, D. Chiu, and G. Agrawal. "A framework for data-intensive computing with cloud bursting." IEEE Cluster computing, 2011.

[17] H.C. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker, "Map-reduce-merge: simplified relational data processing on large clusters." Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007.