Harmony: An Approach for Geo-distributed Processing of Big-Data Applications

Han Zhang
Department of Computer Science
National University of Singapore
Singapore
hanz18@comp.nus.edu.sg

Lavanya Ramapantulu
Computer Science Group
International Institute of Information Technology
Hyderabad, India
lavanya.r@iiit.ac.in

Yong Meng Teo
Department of Computer Science
National University of Singapore
Singapore
teoym@comp.nus.edu.sg

***Abstract*—Big-data application processing is increasingly geo-distributed, a paradigm shift from the traditional cluster-based processing frameworks. As the communication time for data movement across geo-distributed data centers is not a design criterion for traditional cluster-based processing approaches, there are research gaps in the algorithms used for staging and scheduling big-data applications for geo-distributed clusters. We address these gaps by proposing Harmony, an approach consisting of both staging and scheduling strategies to minimize an application's total execution time. The staging strategy of Harmony exploits the intra-stage parallelism by having concurrent operators within a stage in contrast to the traditional Apache spark which uses fine-grained stages, thus reducing the computation time within each stage. Secondly, the scheduling strategy of Harmony reduces data transfers between geo-distributed data centers by exploiting data locality and thus reducing communication time and total execution time. The proposed approach Harmony achieves a speedup of two times with respect to geo-distributed Apache Spark. In addition, Harmony achieves a speedup of 1.6 times and 2.1 times when compared with the state-of-the-art framework Iridium for geo-distributed analytics over five locations with uniform and non-uniform network link bandwidths respectively.**

***Index Terms*—geo-distributed processing, data-centers, performance analysis, scheduling**

I. Introduction

The world's data centers are increasingly becoming geo-distributed by operating data-centers across different geographical locations to provide services for users in different regions [1], [2]. Thus, data generated by user activities or service records is originally distributed geographically. In addition, the continuous growth of IoT devices and sensor networks produce geo-distributed data. Besides geo-distributed data generation, resources such as computing, storage etc. are geo-distributed. Many organizations have data-centers (DCs) or use cloud services in different regions [2], [3]. For example, as of May 2019, Microsoft and Google have DCs in 54 and 16 different geographical locations respectively [4], [5]. By having resources globally these DCs not only improve the quality-of-service in all regions but also mitigate the high latency of cloud resources by lowering the cost of data transfer. Moreover, by having resources all over the world, DCs are more resilient and enhance their ability to recover from disasters.

While there are many research works [6]–[10] on optimizing traditional cluster processing using distributed frameworks such as Apache Hadoop [11] or Spark [12], [13], they are sub-optimal for geo-distributed analytics. As these research works use traditional approaches as a baseline for data placement and task scheduling, they incur higher latency for data movement when executing on geo-distributed systems. In addition, many of these works [6]–[9] support only map and reduce operations, while many recent big-data applications demonstrate the need for a wider set of operators.

While processing geo-distributed data in a timely manner is of great significance, approaches for geo-distributed processing are still in its infancy. One common approach for organizations to process geo-distributed data is copying remote data to a central data-center, which is not scalable and affects network congestion adversely by increasing bandwidth demand and introducing high latency [2]. Hence, current works on optimizing geo-distributed processing mainly focus on reducing data transfer between data-centers and assume infinite computational resources within each data-center [1], [2], [14].

While traditional cluster-based distributed frameworks [6]–[10] are network agnostic, existing geo-distributed processing frameworks [1], [2], [14] are compute capacity agnostic. In contrast, we propose an approach Harmony, that considers both network bandwidth and compute capacity to determine staging and scheduling strategies to minimize total execution time.

Currently big-data application processing is generally modelled as a Directed Acyclic Graph (DAG) [1], [2], which is also widely adopted in the area of high performance computing [15]. However, the time-complexity in scheduling a program DAG on a cluster with multiple processors is NP-Complete [16] and on geo-distributed systems is NP-Hard [1]. Thus, there is a need to design a scheduler with realistic time. This paper proposes an approach Harmony that addresses this need by using a greedy staging strategy to partition a DAG into stages along the critical path length. In addition, Harmony's staging strategy makes use of intra-stage parallelism by grouping operators without shuffle dependency into the same stage. Such a coarse-grained staging uses the compute capacity

978-1-7281-4734-5/19/$31.00

of resources within a data center more effectively in contrast to fine-grained stages of traditional cluster-based approaches, thus reducing both data transfer time and execution-time. In summary, the key contributions of Harmony towards executing big-data applications on geo-distributed systems are:

- a new staging strategy along the DAG's critical path length to exploit intra-stage parallelism and reduce data transfers, by including operators without shuffle dependency in the same stage
- a new task scheduling strategy within each stage using location-aware mapping of tasks to geo-distributed DC systems to reduce execution time.

We evaluate Harmony's staging and scheduling strategies by executing a widely used big-data analytics application, page rank [17] on both traditional cluster-based framework [13] and on a geo-distributed analytics framework, Iridium [2]. We first employ a model to simulate Harmony's staging and scheduling strategies and validate this model with real measurements on a geo-distributed clusters. Next, based on the model we compare the execution time of Harmony with both the frameworks and show that Harmony achieves a speedup of two times and 2.1 times over Spark and Iridium respectively.

The rest of the paper is organized as follows. We discuss the related work in Section II and describe the proposed approach Harmony in Section III. We present our evaluation methodology and results in Section IV and conclude in Section V.

II. Related Work

The state-of-the-art research in the area of executing big-data applications on geo-distributed systems are mainly classified into works (i) considering compute capacities and efficiently processing data in a fault-tolerant manner on distributed systems and (ii) considering network bandwidth to optimize inter-datacenter data transfer times on geo-distributed systems. In this section we compare and contrast the proposed approach Harmony with these two types of research works.

1) Big-data processing considering compute capacities: Big-data processing algorithms and programming models have evolved to overcome the disadvantages of traditional distributed parallel programming models such as MPI [18]. For example, implementing fault-tolerance in the MPI impacts the execution time performance of big-data applications adversely [19]. Distributed programming models such as MapReduce overcome this challenge by implementing fault tolerance as part of the distributed cluster system [20]. However MapReduce supports only two operators, map and reduce thus leaving users to program multiple MapReduce jobs and schedule them in a desired order to implement complex data processing applications. In contrast, Apache Spark [13] improves MapReduce by expanding the number of operators and allowing for intermediate results to be stored in memory to improve the efficiency of large-scale data processing. However, as data generation and computing resources are becoming geo-distributed, such big-data processing frameworks do not perform well as their optimization only consider compute capacities resulting in longer execution times when data needs to be transferred across geo-distributed systems. Compared to these works, Harmony's staging and task scheduling strategies considers both compute capacity and network bandwidth to minimize data transfer times across geo-distributed systems thus resulting in improved total execution time.

2) Big-data processing considering network bandwidth: For geo-distributed big-data processing, many approaches have been proposed to optimize the inter-datacenter transfer time and thus improve total execution time [1], [2], [8], [14], [21]. Afrati et al. [8] proposed Meta-MapReduce to reduce data movement by considering data locality and operator characteristics. Pixida [1] focuses on reducing network traffic in scheduling by formulating the network traffic problem in geo-distributed setting as a min-cut of network flow to optimize inter-datacenter traffic. Ryden et al. proposed Nebula [21] to schedule tasks by estimating overheads using a location-aware scheduler to minimize overall job completion time. Li et al. introduced a Traffic-aware [14] algorithm to minimize the shuffle phase inter-datacenter traffic by solving a joint data and task allocation problem. However most of these works formulate the problem as an integer programming problem to find data-centers with poor network bandwidth and minimize execution time by moving data out of these bottlenecks.

Similarly, Iridium [2] formulates the shuffle problem as a linear programming problem. Iridium's main objective is to minimize query response time in a geo-distributed setting by focusing on data transfers in queries which happen across data-centers. By solving the linear programming problem, Iridium assigns tasks to optimize inter-datacenter data transfer. In contrast to these works, the proposed approach Harmony, considers both computation capacity of the data centers along with data transfer times. Closer to our approach is Flutter [10], which considers both computation time and data transfer time to minimize total execution time. However, we model the program as DAG and use the critical path length for stage and exploit intra-stage parallelism by grouping operators without shuffle dependency to minimize and to mask communication latency.

III. Approach

In this section, we first present an overview of the proposed approach Harmony, followed by a detailed motivational example of how it differs from traditional cluster based approach like Spark. Next, we present the algorithmic details of the two main contributions of Harmony, the staging and scheduling strategies.

A. Overview

The proposed approach Harmony consists of two main steps, staging and scheduling as shown in Figure 1. In the staging step, a given program represented as a DAG is divided into several stages. In contrast to traditional cluster-based approaches, Harmony's staging strategy considers the critical path length to group the operators into stages. Next, the parallel tasks within each stage are scheduled to the different

geo-distributed DCs exploiting both inter-operator and intra-operator parallelism. The key advantage of this approach is that from a user's perspective, the programming model of the distributed program is exactly the same as the geo-distributed program proposed in Harmony. Thus, Harmony not only leverages the fault-tolerance aspects of distributed programming model but also, in contrast to related work, enhances the performance of such a programming model on geo-distributed systems.

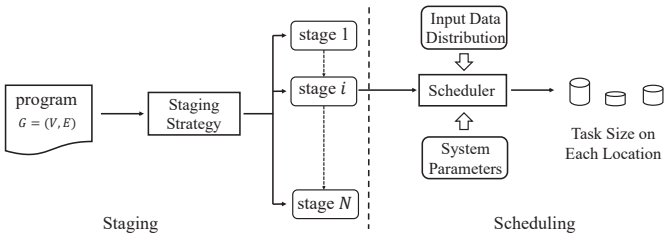


Fig. 1. Overview of Our Approach

Harmony's staging strategy estimates the execution time of each operator on a unit size of data with unlimited resources to determine the critical path of the program. The critical path of the program is the execution of multiple operators in sequence and in the worst case total execution time is limited by the sequential fraction of the program [22]. Thus by dividing the program into multiple stages based on the operators in the critical path, Harmony's staging strategy ensures that the execution time does not exceed the minimum time given infinite resources and limited by the sequential path of the program. Next, as shown in Figure 1, the scheduler uses both input data distribution and system parameters such as available bandwidth to distribute the parallel tasks within each stage across the geo-distributed DCs to minimize both data transfer time and execution time. This task scheduling strategy of Harmony is done in a manner such that all tasks across the geo-distributed DCs complete execution at the same time. Thus, in contrast to related work which do not consider heterogeneity among the geo-distributed DCs, Harmony's task scheduler, assigns tasks based on the compute capacity across the DCs. This is illustrated by the different task sizes on each location as shown in Figure 1. To illustrate these advantages of Harmony over traditional distributed systems approach like Spark, we compare the two approaches using an example below.

B. Motivational Example

To illustrate the need for a new approach for geo-distributed data processing, we present an example program. First a program is represented as a Directed Acyclic Graph (DAG) as shown in Figure 2, where nodes represent operators and edges denote data dependencies. $G = (V, E)$ denotes a program DAG, where V is a set of nodes and E is a set of edges.

Traditional cluster based distributed processing approaches such as Apache Spark use shuffle dependency to group operators into stages. Such a staging strategy uses in-memory

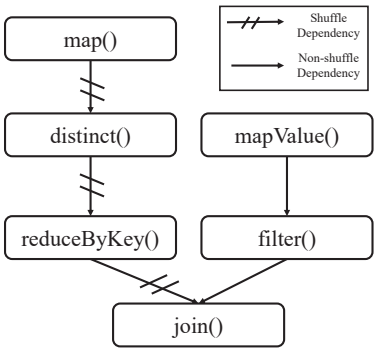
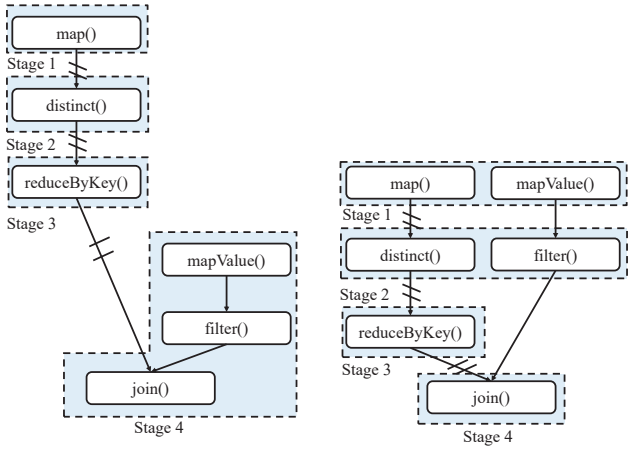


Fig. 2. An Example of a Program DAG

computation by storing intermediate data in memory and the non-memory data is communicated to different nodes for processing. In addition, staging using shuffle dependency ensures that within each stage, operations on different nodes can be performed in parallel without retrieving data from other nodes. An example of Apache Spark's staging strategy is shown in Figure 3(a). However, this staging strategy introduces



(a) Apache Spark's Staging Strategy (b) Harmony's Staging Strategy

Fig. 3. Program DAG with Different Staging Strategies

additional data dependencies into the program DAG, which reduces inter-operator parallelism and leads to extra execution time. In Figure 3(a), mapValue() and map() can be executed in parallel. After staging, mapValue() is assigned in Stage 4 which cannot be executed until all previous are completed. By introducing stages into program DAG as synchronization barriers, Apache Spark prevents some operators from being executed ahead in previous stages. If we assume each operator takes one unit time to execute, Apache Spark's staging strategy shown in Figure 3(a) leads to six units of time to complete while our staging strategy only takes four units of time as shown in Figure 3(b).

To achieve the goal of minimizing total execution time, Harmony's staging strategy focuses on both inter-operator and

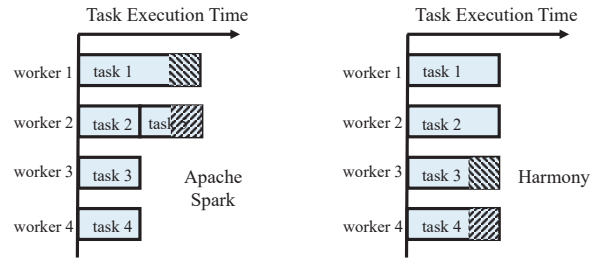
intra-operator parallelism. In the program DAG, Harmony's staging strategy identifies that the optimal execution path is dominated by the critical path, which is the sequence of operators with longest execution time. On the critical path, all operators can only be executed sequentially while other operators can be executed in parallel. Hence, Harmony's staging strategy focuses on the critical path to achieve intra-stage parallelism by grouping non-dependent operators in the same stage to minimize total execution time. In addition, the proposed staging strategy groups operators according to shuffle dependencies along the critical path to reduce data transfer and thus accelerate processing time.

Once operators are grouped in stages, Harmony's next step is dividing stages into tasks and scheduling them on geo-distributed DCs. In Apache Spark, this is done in two separate steps. The first step divides each stage into tasks according to the number of data partitions. Then tasks are scheduled by Apache Spark's task scheduler with the default round-robin scheduling algorithm. To divide stages into tasks, Apache Spark only considers the number of data partitions but not on the compute capacity of the processing nodes. Thus on clusters that are homogeneous systems, based on the number of data partitions, each task from the same stage takes almost the same time to execute.

However, in a geo-distributed data processing setup, the DCs are more likely to be heterogeneous systems, wherein same task size will potentially lead to differing execution time. In an extreme case, if the processing capacity varies a lot, the slowest worker may take up to ten times more time than on average. Another disadvantage of Apache Spark is determining the number of tasks without considering the number of workers may lead to multiple rounds of scheduling to complete task execution within one stage. As shown in Figure 4(a), Apache Spark scheduler divides the stages into five tasks even though there are only four worker nodes in contrast to Harmony's scheduler which will divide the stages into only four tasks as shown in Figure 4(b). Similarly, given the heterogeneity of the system, equally divided task sizes will take different execution times on different nodes with varying computational capacity, whereas Harmony's scheduler divides the tasks per worker node considering the computational capacity such that all tasks complete execution at the same time as shown in Figure 4(b).

C. A Greedy Staging Strategy Based on Critical Path

In this subsection, we present our staging strategy. Generally, a program can be considered as a set of operators on a set of data-sets. If we take operators as nodes and dataflow as edges, a program can be represented as a graph. As a program may contain conditional loops, every program may not be represented as a DAG. However, in distributed setting, most of the programming models do not support conditional loops and loop length is known a priori and can be easily unrolled. Hence, in the setting of big-data applications executing on geo-distributed systems, in this paper we focus on programs which can be represented as a DAG.



(a) Apache Spark's Scheduling (b) Harmony's Scheduling

Fig. 4. Two Drawbacks of Equal Data Partition

The complexity of scheduling tasks from a DAG on a multi-processor system is a NP-Complete problem [16]. In a geo-distributed setting, considering communication time and computation capacity increases the time complexity further. Thus, we propose a greedy approach to address this challenge. To simplify the DAG scheduling problem, we first try to reduce the complexity by grouping operators into stages. Harmony's staging strategy first computes the critical path by determining the set of operator nodes that gives the shortest time to execute the DAG as shown in Algorithm 1. The time complexity of Algorithm 1 is $O(v^2)$.

Input: $G = (V, E)$: program DAG
Output: CriticalPath : a set of nodes which are on the critical path
V.append(s, t);
for *each node* v *in* $V - \{s, t\}$ **do**
  dist$[v] = -\infty$;
  if *(v.inDegree* $== 0$*)* **then**
    Add edge to E (s, v);
  end
  if *(v.outDegree* $== 0$*)* **then**
    Add edge to E (v, t);
  end
end
TopologySort(G);
for *each node* v *in* $V - S$ **do**
  for *each node* u *where* $(u, v) \in E$ **do**
    if *(dist[v]* $<$ *dist[u]+v.weight)* **then**
      dist$[v]$ = dist$[u]$+v.weight;
      Path$[v]$ = u;
    end
  end
end
v = t;
while *Path[v]<>* s **do**
  CriticalPath.append(Path$[v]$);
  v =Path$[v]$;
end
return CriticalPath;

Algorithm 1: IdentifyCriticalPath()

Once the critical path is identified, our greedy algorithm estimates the starting and ending time for all operators in the program DAG as shown in Algorithm 2.

```
Input: G = (V, E) : program DAG
Output: G[V, E] : program DAG with assigned v.start
        and v.end in V
currentTime = 0;
for i = 1 to MaxTopologyOrder do
    temp = 0; for all nodes where v.TopologyOrder =
     i do
        v.start = currentTime;
        v.end = v.start + v.weight;
        if temp < v.end then
            temp = v.end;
        end
    end
    currentTime+ =temp;
end
```

Algorithm 2: AssignStartEndTime()

After detecting the critical path, Harmony's staging strategy divides this critical path into different stages. The algorithm for the staging strategy is described in Algorithm 3.

```
Input: G = (V, E) : program DAG
Output: S[] : a list of stages, each element is the set
        of nodes belong to this stage
CP = IdentifyCriticalPath(G);
AssignStartEndTime(G);
stageNo = 0;
for each node v in CP with TopologyOrder do
    v.stage = stageNo;
    S[stageNo].append(v);
    if (v.dependencyType = 'Shuffle') then
        stageNo++;
    end
    stage[v.stage].end = v.end;
end
return S[];
```

Algorithm 3: StagingStrategyCP()

Once the critical path nodes from the graph are identified and grouped in the corresponding stages, the remaining nodes of the graph are grouped into their respective stages. However, as stages are divided according to shuffle dependencies, there may be operators whose ending time is later than the ending time of its corresponding stage. To overcome this problem, we split such operators into two parts, wherein one part is within this stage and the rest is assigned to the next stage, thus exploiting intra-operator parallelism. This strategy for staging the non-critical path nodes of the DAG is shown in Algorithm 4.

In summary, Harmony's greedy staging strategy firstly tries to compute the starting and ending time for all operators, followed by detecting the critical path using topological sorting and dynamic programming. In addition, the operators are assigned stages based on shuffle dependency and for nodes that are not on the critical path, their start and end times are considered to assign them to the corresponding stages exploiting both intra-stage and intra-operator parallelism.

```
Input: G = (V, E) : program DAG
Output: S[] : a list of stages, each element is the set
        of nodes belong to this stage
CP = IdentifyCriticalPath(G);
AssignStartEndTime(G);
for all nodes v not in CP do
    for all stages s in stage[] do
        if (s.end ≤ v.end) then
            v.stage = s.stageNo;
            S[stageNo].append(v);
            break;
        end
        if (v.start ≤ s.end) and (v.end > s.end) then
            v.stage = s.stageNo;
            add new node v' to G;
            v'.start = s.end;
            v'.end = v.end;
            v.end = s.end;
            S[stageNo].append(v');
            break;
        end
    end
end
return S[];
```

Algorithm 4: StagingStrategyNonCP()

D. Scheduling Tasks onto a Geo-distributed System

Once the program has been divided into stages, the next step is to divide each stage into tasks and to schedule these tasks. In traditional distributed frameworks such as Apache Spark, this step is further divided into two sub steps. Firstly, Apache Spark divides the stages into tasks based on data partitioning without considering the number of workers or the compute capacity of worker nodes. Next, the scheduler of Apache Spark uses the round-robin algorithm to map these tasks onto the worker nodes without considering the geo-distributed nature and the available bandwidth of each DC. In contrast, Harmony's scheduling strategy considers computational capacity, heterogeneity of the DC, number of worker nodes and available network bandwidth at each DC before dividing the stages into tasks.

Harmony's scheduling algorithm uses only a single step to both divide the stages into tasks and mapping of the tasks onto the geo-distributed DC systems. Using the information of the input data to be processed and the computational capacity at each location, the amount of data to be processed at each geo-distributed system which is the task size, $D^{'}_{i,j}$ on location j for stage i is computed using Algorithm 5. The aim of this algorithm is to minimize the total execution time which in turn

depends on the time taken by the critical path denoted by:

$$T_i^c = max(T_{i,j}^c) \quad (1)$$

Input: i : current stage
$D_{i,j}$: input data on location j in stage i
C_j : the computation capacity of location j
Output: $D_i^{'}$[] : the size of task on each location in stage i
Data $= 0$;
Compute $= 0$;
for $j = 1$ *to* N **do**
  Data $+ = D_{i,j}$;
  Compute $+ = C_j$;
end
t = Data / Compute;
for $j = 1$ *to* N **do**
  $D_{i,j}^{'}$ = t $\times C_j$;
end
return $D_i^{'}$[];
Algorithm 5: ComputeDataSizeForStage()

To minimize total computation time, firstly, we try to find the minimal execution time of each stage. As the total data size and total computation capacity is known, the total execution time for stage i can be computed as below, where D_i is the total input size of stage i and C_j is the computation capacity of location j.

$$T_i^c = \frac{D_i}{\sum_{j=1}^{N} C_j} \quad (2)$$

To reach this boundary, we re-allocate data in different locations as shown in Algorithm 5. After deciding the data distribution between locations, we compute the plan for data transfer. In our approach, we assume the network model as a peer-to-peer network, where each location has both up-link and down-link with all other locations. An example of a peer-to-peer network is shown in Figure 5. We denote the network model by a directed graph $H = (L, P)$, where L denotes the locations and P denotes the link between locations with a weight representing the data transfer cost.

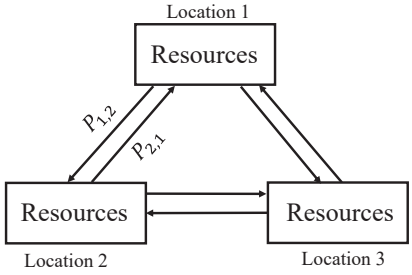


Fig. 5. Peer-to-peer Network Model on Cloud Locations

When re-allocating the input data, we aim to minimize the data transfer cost by sending data via the cheapest link to the locations which needs data. The data transfer strategy is shown in Algorithm 6.

Input: i : current stage
$D_{i,j}$: original input data on location j in stage i
$D_{i,j}^{'}$: final input data on location j in stage i
$H = (L, P)$: Link graph of the geo-distributed system
Output: Traffic[] : a list of (source, destination, size of data transfer)
for $j = 1$ *to* N **do**
  if $D_{i,j}^{'} < D_{i,j}$ **then**
    DataToTransfer $= D_{i,j} - D_{i,j}^{'}$;
    Sort(P from location j);
    for $k =$ *all other locations in order* **do**
      DataNeeded = $D_{i,j}^{'} - D_{i,j}$;
      if *DataToTransfer* $>$ *DataNeeded* **then**
        Traffic.append([j,k,DataNeeded])
        DataToTransfer $-=$ Dataneed;
      end
      if *DataToTransfer* $\leq$ *DataNeeded* **then**
        Traffic.append([j, k, DataToTransfer]);
        break;
      end
    end
  end
end
return Traffic[];
Algorithm 6: DataTransferAlgorithm()

Once data has been re-allocated on all locations, the next step is to assign tasks to each location according to the data and computation capacity for parallel execution.

IV. Evaluation

In this section, we first present an analytical model for analyzing geo-distributed data processing, followed by validation of our model with two different popular big data analytics workloads using measurement data by executing a geo-distributed version of Apache Spark across geo-distributed Amazon EC2 instances. Next, we evaluate Harmony's execution time performance with both Apache Spark and Iridium.

A. Analytical Model and Validation

The analytical model presented here determines the total execution time of a given big data application represented as a DAG executing on a known geo-distributed system. The notation used are listed in Table I. We also validated our model on two different popular big-data applications, WordCount and PageRank on geo-distributed AWS EC2 clusters.

In our analytical model, a program is represented as a DAG $G = (V, E)$ with i sequential stages. Given a program is divided into i sequential stages, the total program execution time T is

$$T = \sum_{i=1}^{N} T_i \quad (3)$$

TABLE I
NOTATION OF THE MODEL

Symbol	Meaning
	Application
$G = (V, E)$	Program DAG where V denotes the set of operators and E denotes data dependency
N	Number of Stages on the critical path
D_i	Total size of input data in stage i
$D_{i,j}$	Size of input data on location j in stage i
ζ_i	Processing time for unit size of data per computation capacity of stage i
	System
C_j	Computation capacity of location j
$B_{j,k}$	Bandwidth of link from location j to k
	Framework
$D^{'}_{i,j}$	Size of task on location j in stage i
T_i	Total execution time for stage i
T^c_i	Computation time of stage i
T^t_i	Communication time of stage i
$R^i_{j,k}$	Re-allocated data from location j to k
$S^i_{j,k}$	Shuffle data from location j to k
	Model Output
T	Total execution time

where T_i is the execution time for each stage and N is the number of stages on the critical path. For each stage, the execution time is

$$T_i = max\big(T^c_i, T^t_i\big) \tag{4}$$

where T^c_i is computation time and T^t_i is communication time for stage i. Here, we assume that the computation and communication can happen concurrently. In other words, once any small chunk of data is ready, it can be directly transferred to other locations.

The computation time for stage i is

$$T^c_i = max\left(\zeta_i \times \frac{D^{'}_{i,j}}{C_j}\right) \tag{5}$$

where $D^{'}_{i,j}$ is the task size of stage i on location j, C_j is the computation capacity (number of executors) at location j and ζ_i is processing time for unit size of data (sec/byte) in stage i for each executor. Here, $D^{'}_{i,j}$ is the scheduling algorithm output of any given geo-distributed processing approach, e.g. Iridium or Harmony. The number of executors on location j, C_j, is a geo-distributed system parameter which is known prior to execution or can be measured. ζ_i is computed based on stage execution time measured on a given data set with one executor. For Harmony, in evaluating scheduling part, the ζ_i is measured and is the same as in Apache Spark. In evaluating staging strategy part, since we cannot directly measure the execution time of the whole stage due to different staging strategy, we measure the execution time of each operator.

TABLE II
BANDWIDTH[MBPS] BETWEEN DIFFERENT LOCATIONS

Bandwidth	Tokyo	N.Virginia	Singapore	Ohio
Tokyo	-	32.4	51.2	47.2
N.Virginia	27.5	-	24.1	247
Singapore	75.0	33.8	-	37.6
Ohio	68.2	201	28.8	-

The communication time for stage i is

$$T^t_i = max\left(\frac{R^i_{j,k} + S^i_{j,k}}{B_{j,k}}\right) \tag{6}$$

where $R^i_{j,k}$ is the size of reallocated data of stage i from location j to k at the beginning of executing, $S^i_{j,k}$ is the size of shuffled data of stage i from location j to k when executing and $B_{j,k}$ is the network bandwidth between location j and k. Here, $R^i_{j,k}$ and $S^i_{j,k}$ are the scheduling algorithm output of given geo-distributed processing approach. The network bandwidth $B_{j,k}$ is the geo-distributed system parameter which can be measured.

To validate our model, we use the WordCount and the PageRank program from Apache Spark [13]. WordCount includes a join operator performs a Map-Reduce string processing with a join operator to show the meaning of each word. PageRank represents an iterative graph processing algorithm which is widely used in big-data processing. PageRank input data is generated by BigDataBench [17] based on Google Web Graph [23]. Applying our staging strategy, the WordCount with join is partitioned into three stages and PageRank program DAG is partitioned into six stages as shown in Figure 6 and 7. Our geo-distributed system consists of Amazon EC2 [24] t2.xlarge instances distributed over four locations as shown in Figure 8. Apache Spark is installed on each instance with HDFS as the distributed file system. Table II shows the bandwidths across different regions, where the result is the average of three measurements.

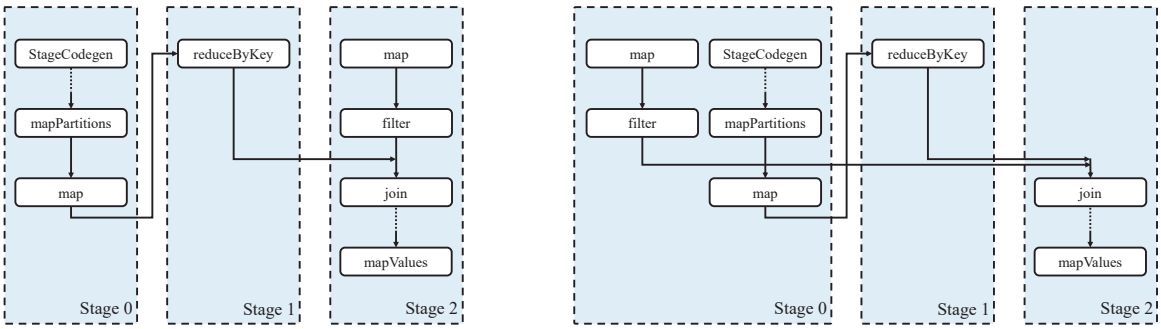
We compute the percentage error between our model prediction and actual measured value running geo-distributed Apache Spark. Table III and IV shows the result of our validation.

TABLE III
MODEL ERROR WITH GEO-DISTRIBUTED APACHE SPARK ON WORDCOUNT WITH JOIN

Input Size	Apache Spark	Model	Error
2.5 GB	297s	329s	10.8%
5 GB	609s	685s	12.5%

B. Performance Evaluation of Harmony with Apache Spark

As we have validated our geo-distributed processing model, in this subsection, we use this model to compare with Apache Spark. We first compare Harmony's staging strategy with Geo-distributed Apache Spark on WordCount with Join. Then, we compare Harmony's scheduling algorithm with Geo-distributed Apache Spark on PageRank.



(a) Apache Spark's Staging Strategy

(b) Harmony's Staging Strategy

Fig. 6. Staging Strategies on WordCount with Join

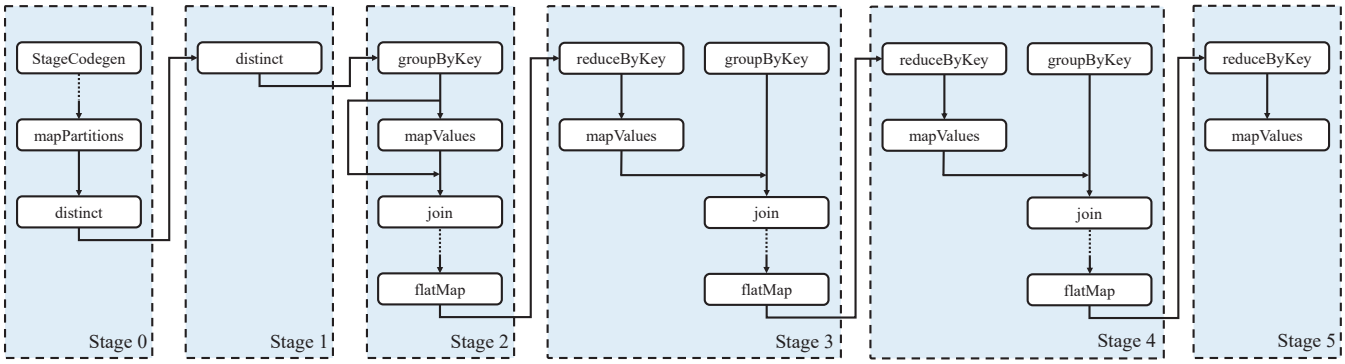


Fig. 7. Proposed Staging Strategy on PageRank

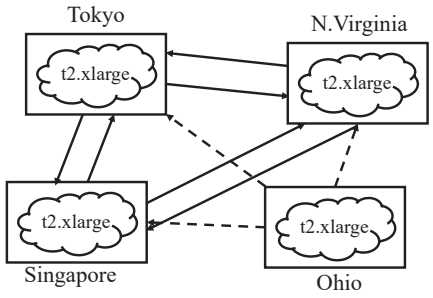


Fig. 8. Geo-distributed Cloud System with Four Locations

The system used in the validation has the same network topology with the previous section. Ohio is the master node and worker nodes are in three regions: Tokyo, N.Virginia and Singapore. The bandwidth between different regions is the same as what is shown in Table II.

To validate Harmony's staging strategy, we set all locations with the same computation capacity. Hence, the scheduling result for Apache Spark and Harmony are the same. We use the performance model to compute the execution time for Apache Spark and Harmony on application WordCount with Join. As shown in figure 9, Harmony's stage 2 execution time is shorter than geo-distributed Apache Spark due to executing first two operators in stage 2 ahead which brings 15% improvement on the total execution time.

Then the computation capacity in Tokyo and N.Virginia is set at 3 times of Singapore to validate Harmony's scheduling algorithm. We compute the total execution time of Apache Spark and Harmony on PageRank. As shown in Figure 10, Apache Spark's total execution time is at more than twice of Harmony's. The completion time of tasks assigned in Singa-

TABLE IV
MODEL ERROR WITH GEO-DISTRIBUTED APACHE SPARK ON PAGERANK

Graph Size		Apache Spark	Model	Error
Nodes	Edges			
4,194,304	21,383,292	154s	162s	5.2%
8,388,608	46,053,197	331s	352s	7.0%
17,777,216	99,184,770	676s	727s	7.0%

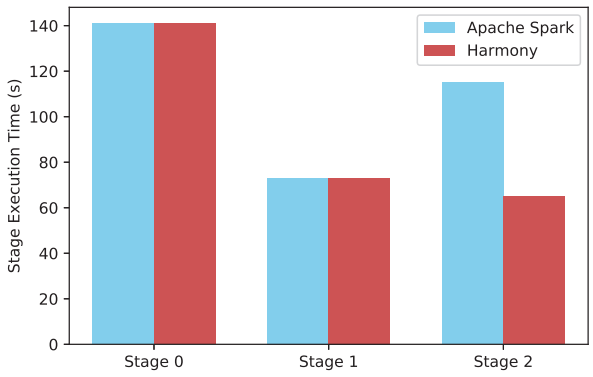


Fig. 9. WordCount with Join Execution Time of Each Stage of Harmony and Apache Spark

pore is the bottleneck because the performance of executors in this region is lower comparing with other regions. However, Apache Spark only assigns tasks considering the number of cores and threads, which leads to longer completion time in the Singapore region.

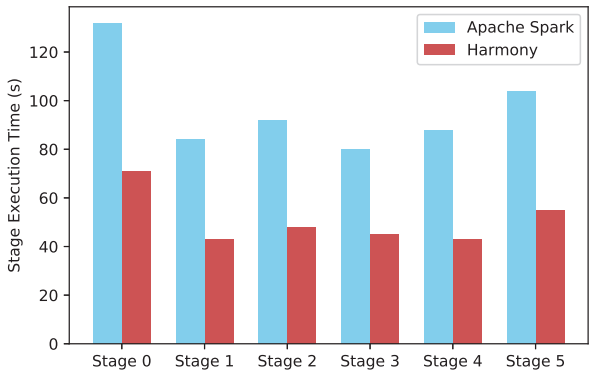


Fig. 10. PageRank Execution Time of Each Stage of Harmony and Apache Spark

Figure 11 shows how stage 0 is executed on geo-distributed system. When Harmony schedules tasks, it consider the computation capacity in each region to minimize the execution time of each stage. Hence, data is moved out of the Singapore region, to balance the processing power between regions. By reducing tasks assigned to Singapore, execution time of each stage is minimized and finished around the same time.

C. Performance Evaluation of Harmony with Iridium

In this section, we compare our approach with Iridium [2] to study the effectiveness of our approach in geo-distributed data processing. As Iridium assumes negligible computation time and only considers communication time, its total execution time is determined by the communication time.

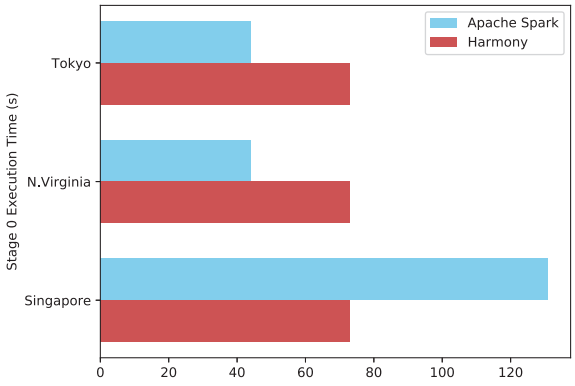


Fig. 11. PageRank Execution Time of Stage 0 on Each Location of Harmony and Apache Spark

In Iridium, the network model is a congestion-free core network compares with our peer-to-peer network model as shown in Figure 12. To model the core network of Iridium

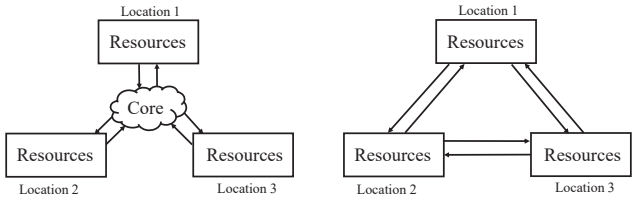


Fig. 12. Core vs Peer-to-Peer Network

with our peer-to-peer network in a fair manner, we consider two cases. Firstly we use a fixed network capacity with uniform links across all the peers (Harmony) and for all the links between the core and the geo-distributed DC (Iridium). Secondly, links have variable bandwidth, but the total capacity of the core network is fixed.

Using these two cases, we compare the execution time of a single stage for both Harmony and Iridium across different number of locations and the results are shown in Figures 13 and 14 for the uniform and non-uniform link bandwidths respectively. As observed from the results, Harmony outperforms Iridium when the number of locations is three and above for both types of networks with uniform and non-uniform link bandwidths. While Iridium incurs a linear increase in execution time due to the contention for the shared resources (network core), the performance impact of increasing the number of locations using Harmony's scheduler is negligible especially after the number of geo-distributed locations is five and above. Even with five data locations, Harmony achieves a speedup of 1.6 times over Iridium when the link bandwidth is uniform and a speed up of 2.1 times when the link bandwidth is non-uniform. Thus, Harmony is better suited to compute data intensive applications on large number of geographically distributed DCs.

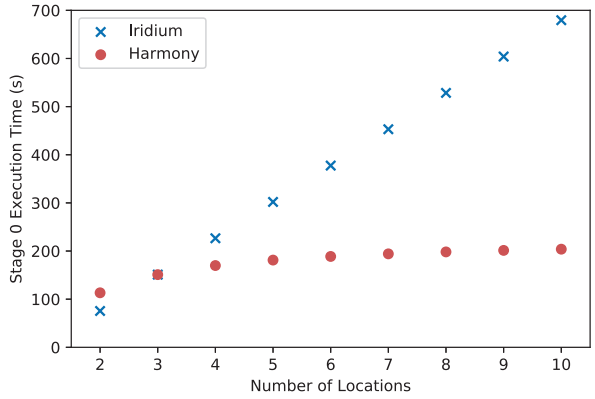


Fig. 13. Core vs Peer-to-Peer Network (Uniform)

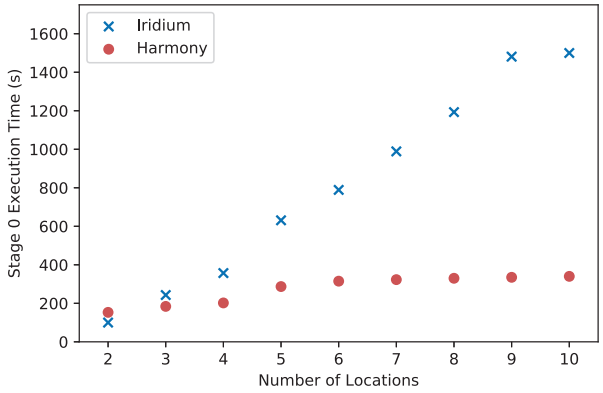


Fig. 14. Core vs Peer-to-Peer Network (Non-Uniform)

V. Conclusion

Compute systems are increasingly becoming geo-distributed for executing data-intensive applications due to both data availability and system resiliency. Given this paradigm shift, traditional distributed frameworks are no longer efficient for data processing on geo-distributed DCs as they do not consider network bandwidth in their task scheduling algorithms. Thus there is a need for a new approach to execute data-intensive applications on geo-distributed systems in a more efficient manner. The challenge in this research problem is the multi-variable optimization becoming computationally hard considering number of worker nodes, compute capacity of each node, network bandwidth availability at each node and data locality of the program including data dependencies. We address this research challenge by presenting a novel approach Harmony, that uses a greedy algorithm focusing on the critical path length to divide program into stages and then scheduling tasks according to estimated minimal execution time considering computation capacity and data transfer cost.

To compare our approach with existing geo-distributed solutions, we propose a geo-distributed processing analytical model and validate it using measurements for two popular applications WordCount and PageRank executing on geo-distributed Amazon EC2 instances. We use the model to evaluate the performance of the proposed approach Harmony with both Apache Spark and state-of-the-art geo-distributed framework Iridium. The evaluation results show that Harmony achieves a speedup of two times with respect geo-distributed Apache Spark. In addition, Harmony achieves a speedup of 1.6 times and 2.1 times when compared with the state-of-the-art framework Iridium for geo-distributed analytics over five locations with uniform and non-uniform network link bandwidths respectively.

References

[1] K. Kloudas, M. Mamede, N. Preguiça, and R. Rodrigues, "Pixida: optimizing data parallel jobs in wide-area data analytics," *Proceedings of the VLDB Endowment*, vol. 9, no. 2, pp. 72–83, 2015.

[2] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4. ACM, 2015, pp. 421–434.

[3] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services," 2010.

[4] Google, "Google datacenter locations," *URL http://www. google. com/about/datacenters/inside/locations/*, 2019.

[5] Microsoft, "Microsoft datacenters," *URL https://azure.microsoft.com/en-us/*, 2019.

[6] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-hadoop: Mapreduce across distributed data centers for data-intensive computing," *Future Generation Computer Systems*, vol. 29, no. 3, pp. 739–750, 2013.

[7] Y. Luo and B. Plale, "Hierarchical mapreduce programming model and scheduling algorithms," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 769–774.

[8] F. Afrati, S. Dolev, S. Sharma, and J. D. Ullman, "Meta-mapreduce: A technique for reducing communication in mapreduce computations," *arXiv preprint arXiv:1508.01171*, 2015.

[9] C. Jayalath, J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running mapreduce across datacenters," *IEEE Transactions on Computers*, p. 1, 2013.

[10] Z. Hu, B. Li, and J. Luo, "Flutter: Scheduling tasks closer to data across geo-distributed datacenters," in *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*. IEEE, 2016, pp. 1–9.

[11] A. Hadoop, "Hadoop," 2009.

[12] 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.

[13] A. Spark, "Apache spark: Lightning-fast cluster computing," *URL http://spark. apache. org*, 2016.

[14] P. Li, S. Guo, T. Miyazaki, X. Liao, H. Jin, A. Y. Zomaya, and K. Wang, "Traffic-aware geo-distributed big data analytics with predictable job completion time," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1785–1796, 2017.

[15] C. Augonnet and R. Namyst, "A unified runtime system for heterogeneous multi-core architectures," in *European Conference on Parallel Processing*. Springer, 2008, pp. 174–183.

[16] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951–967, 1994.

[17] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *et al.*, "Bigdatabench: A big data benchmark suite from internet services," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 488–499.

[18] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2004, pp. 97–104.

[19] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, "A survey on geographically distributed big-data processing using mapreduce," *arXiv preprint arXiv:1707.01869*, 2017.

[20] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[21] M. Ryden, K. Oh, A. Chandra, and J. Weissman, "Nebula: Distributed edge cloud for data-intensive computing," in *Collaboration Technologies and Systems (CTS), 2014 International Conference on*. IEEE, 2014, pp. 491–492.

[22] S. Eyerman and L. Eeckhout, "Modeling critical sections in amdahl's law and its implications for multicore design," in *ISCA*, 2010.

[23] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[24] A. E. C. Cloud, "Amazon web services," *Retrieved November*, vol. 9, p. 2011, 2011.