2014 IEEE 28th International Parallel & Distributed Processing Symposium

Pythia: Faster Big Data in Motion through Predictive Software-Defined Network Optimization at Runtime

Marcelo Veiga Neves, César A.F. De Rose
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre, Brazil
Email: marcelo.neves@acad.pucrs.br

Kostas Katrinis
IBM Research – Ireland
Dublin, Ireland
Email: katrinisk@ie.ibm.com

Hubertus Franke
IBM TJ Watson Research Center
Yorktown Heights, NY, US
Email: frankeh@us.ibm.com

***Abstract*—The rise of Internet of Things sensors, social networking and mobile devices has led to an explosion of available data. Gaining insights into this data has led to the area of Big Data analytics. The MapReduce framework, as implemented in Hadoop, is one of the most popular frameworks for Big Data analysis. To handle the ever-increasing data size, Hadoop is a scalable framework that allows dedicated, seemingly unbound numbers of servers to participate in the analytics process. Response time of an analytics request is an important factor for time to value/insights. While the compute and disk I/O requirements can be scaled with the number of servers, scaling the system leads to increased network traffic. Arguably, the communication-heavy phase of MapReduce contributes significantly to the overall response time; the problem is further aggravated, if communication patterns are heavily skewed, as is not uncommon in many MapReduce workloads. In this paper we present a system that reduces the skew impact by transparently predicting data communication volume at runtime and mapping the many end-to-end flows among the various processes to the underlying network, using emerging software-defined networking technologies to avoid hotspots in the network. Dependent on the network oversubscription ratio , we *demonstrate reduction in job completion time* between 3% and 46% for popular MapReduce benchmarks like Sort and Nutch.**

***Keywords*-Distributed computing; Data communication; Data processing;**

I. Introduction

Driven by the tremendous adoption of electronic devices and the high penetration of broadband connectivity globally, the generation of electronic data grows at an unprecedented rate. In fact, this rate is expected to steadily grow due to increasing adoption of trending data-heavy technologies, arguably Internet of Things, social networking and mobile computing. The knowledge that can be extracted by processing this vast amount of data has sparked interest in building scalable, commodity-hardware based and easy to program systems, resulting today in a significant number of purpose-built data-intensive analytics frameworks (e.g., MapReduce [1], Dryad [2] and IBM Infosphere Streams [3]), often captured by the market-coined term "Big Data" analytics.

The data-heavy nature of workloads run by Big Data analytics systems, in conjunction with the need to scale-out to hundreds or even thousands of compute nodes for capacity (speedup) or capability (immense input/scratch storage of the workload requiring proportionally high number of nodes) reasons, incurs high data-movement activity in the datacenter, where such analytics systems are deployed. For instance, a recent analysis of MapReduce (MR) traces from Facebook revealed that 33% of the execution time of a large number of jobs is spent at the MapReduce phase that shuffles data between the various data-crunching nodes [4]. This creates an obvious incentive to optimize the communication-intensive part of such applications via appropriate dynamic network control.

Until recently, the toolset for application-induced network control was limited to a small set of protocols (e.g., Quality of Service protocols) that were embedded into network devices at manufacturing time, unable to be dynamically changed to keep up with application needs. Software-Defined Networks (SDN) break this inflexibility, offering fine-grained programmability of the network and high modularity to allow for directly interfacing application orchestration systems with the network. Leveraging from this evolution in networking technology, this paper presents Pythia[1], a system that employs real-time communication intent prediction for Hadoop [5] and uses this predictive knowledge to optimize the datacenter network at runtime, aiming at Hadoop MapReduce acceleration.

More specifically, Pythia sports an instrumentation middleware that tracks - transparently to applications and the Hadoop framework itself - Hadoop map task processes at runtime and mines intermediate output data that will eventually be transferred to remote nodes for further processing during the reduction phase. Prediction intelligence is collected at a central controller and in turn ingested by a chain of network control algorithms (routing, flow scheduling) that optimize network resource allocation against faster Hadoop MapReduce shuffle phase completion and thus accelerate MapReduce job completion.

We prototyped the proposed scheme in a high-end cluster

[1]According to Greek Mythology, Pythia was an ancient Greek priestess at the Oracle in Delphi, widely credited for her prophecies inspired by Apollo.

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.20

and evaluated the performance improvement it brings to various HiBench [6] benchmarks. Our results manifest that Pythia achieves consistent acceleration of MapReduce workloads, with speedup ranging from 3% to 46%, depending on the network blocking ratio and the workload. We also present results confirming that our communication intent prediction approach and implementation is able to accurately (in terms of shuffle flow size) and timely (within several seconds prior to actual flow occurrence in the network) forecast MapReduce intermediate data movement.

The direct value driven by this work is in the performance improvement brought to Hadoop MapReduce by optimizing its communication-intensive phase via application-specific instrumentation and specialized network control. Beyond this immediate contribution, a more far-reaching anticipated impact of the present work lies in forming a concrete and tangible materialization of the value that the software-defined concept can bring to the end-user by enabling tighter and constant application/workload affinity to the IT infrastructure.

The rest of this paper is structured as follows. We review and put our work in the context of related work in the Section VI. Section II builds up on background concepts and technologies used in this paper and also drives the motivation for this work by uncovering open problems and discussing value potential. Sections III and IV present the Hadoop-specific and network-related technical specification of our system. We present a quantitative evaluation of the benefit of our system to Hadoop MapReduce workloads in Section V and conclude with summary and elevation of the findings in Section VII.

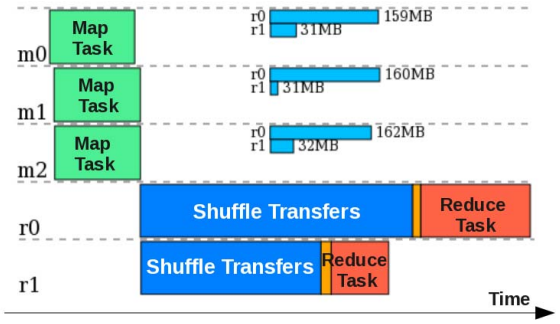
II. Background and Motivation

Hadoop MapReduce [5] is the open-source realization of Google's MapReduce [1] distributed data-processing framework. It is gaining increasing traction and deployment base in various domains requiring batch-processing large-scale analytics, primarily due to linear system scalability, support for both structured and unstructured data sources, a fairly straightforward programming model and transparent to the user runtime parallelism. A remarkably versatile set of applications can be implemented by using the two primitive functions of the MR model, namely *map* and *reduce*. The map function ingests input data records (values) and maps them to an application-specific key-space. In turn, the output ("intermediate map output" in Hadoop terminology) of the map function is sorted and fed to an application-specific reduce function (e.g., summation of numeric values). From an implementation perspective, Hadoop job execution is orchestrated by a two-level hierarchy of control entities: the "jobtracker", which runs on the Hadoop cluster's master node and is the job-level control entity, and one or more "tasktrackers", each running on every Hadoop compute node (termed "slave").

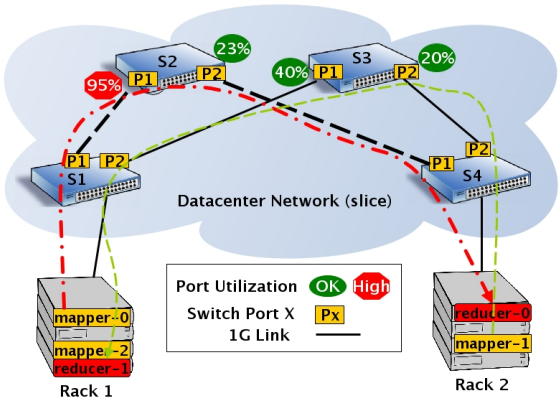
Among others, a tasktracker's role is to initiate/control map/reduce task processes that implement the application-provided map/reduce function. Specifically, a tasktracker starts a map task with a discrete chunk of data (typically a distributed file system data block) as input; the sorted intermediate output of the map task is in turn hashed to the set of reducers (which are also started by tasktrackers across the Hadoop cluster), making sure that no two reducers process <key,value> pairs with the same key. Following coordination steps taken between the tasktrackers and the jobtracker, the tasktracker that a sample reducer is controlled by fetches from each finished map task's tasktracker the set of <key,value> pairs that hash to the specific reducer-ID. Seen from the map task's tasktracker server, the communication pattern corresponds to a "shuffling" of intermediate output data from the mapper server to the job's reducer servers, thereby being termed as the "shuffle" phase. After a reducer task has fetched all data produced by the entire set of map tasks, it proceeds to the final reduce phase and then writes back the reduction result to a file stored in the distributed file system. Given the increasingly high volumes of data that typical analytics applications ingest, it is straightforward that the volume of data that need to be shuffled during a Hadoop job is in many cases relatively high, thereby calling for solutions to shorten the duration of the shuffle phase. In the rest of this section, we motivate this through a MapReduce job execution example.

Figure 1a depicts the sequence diagram of the execution of a toy-sized sort job in a 1Gbps non-blocking network, obtained by a custom visualization tool we have developed. The job uses three map tasks (slots) and two reducers, whereby the three distinct phases of interest in this paper are clearly annotated (distributed file system phases are omitted for brevity). It can be clearly observed that the network-heavy shuffle phase takes up a substantial fraction of job execution time, motivating thus further work to optimize the network against the network-throughput intensive phase of MapReduce. An additional observation that also motivates the type of work presented in this paper is the disproportionality of the intermediate output data sizes fetched by the two reducers; specifically, reducer-0 receives 5x times more data compared to reducer-1. This is not an uncommon problem in MapReduce executions ("job skew" effect [7]) caused by non-uniform data distribution in the key space. While this problem can be addressed at multiple levels (e.g., by dynamically adapting the partitioning function that governs the volume of data assigned to each reducer), our work intends to address it at the system/network level. Intuitively, if reducer-0 receives five times more data, then it is straightforward that also the flows terminated at reducer-0 should get five times more network capacity (bandwidth) than reducer-1. It is this end goal that motivates the application-aware, flow-level network control materialized in this paper.

Until recently, the network in commodity deployments



(a) Hadoop sort job sequence diagram



(b) Adversarial shuffle flow allocation to the network

Figure 1: Motivational Hadoop job analysis and implications of conventional network control

was from a control/management point operated as a black-box, offering very low capability of application-induced, fine-grained control (e.g., controlling network policy at the granularity of a single flow). Software-defined networks (SDN) materialize the long-awaited decoupling between the control and data forwarding logic of network elements (switches/routers), moving the control-plane off the network elements and on to a centralized network controller, where virtually any logic controlling network elements in any desired ways can be implemented in software. In the context of Big Data applications, software-defined networks provide for the ability to program the network at runtime in a manner, such that data movement is optimized against faster, service-aware and more resilient application execution.

Still, the transition to an application-controlled software-defined infrastructure is not straightforward. We make the case for this in Figure 1b, where we depict a candidate execution of the job shown in Figure 1a on two racks within a wider datacenter. In this example, the network has two alternative paths between the two racks (Path-1 and Path-2 in Figure 1b) and employs Equal Cost Multi-Pathing (ECMP [8]) for flow allocation to multiple paths. ECMP has been proposed as a solution in such environments [9], mainly due to its simplicity and efficient implementation on network hardware. Figure 1b shows also utilization (buffer occupancy) at switch ports facing the datacenter network. Given that ECMP employs random local hashing of flow packets to output ports at every network switch, a possible allocation of two shuffle flows - namely reducer-0 fetching <key,value> pairs from mapper-0 (flow-1) and reducer-1 fetching <key,value> pairs from mapper-1 (flow-2) - is shown. Due to the load-unawareness of ECMP-like flow allocation, this candidate allocation leads to the adversarial effect of assigning a relatively large flow (159MB) to a highly-loaded path (95% load, Path-1), even if there is available network capacity to complete the shuffle transfer faster. Note that this effect is not a side-effect of nominal network capacity, i.e., bad flow packing can lead to sub-optimal network utilization even in non-blocking networks [10]. Replacing ECMP with a load-aware flow scheduling scheme (e.g., Hedera [10]) would to some extent avoid such adversarial flow allocations, however still not manage to unleash the entire optimization potential. For instance, in the example presented in Figure 1, schemes like Hedera would fail to recognize the criticality of flow-1 to MapReduce job progress and as such, even if both flows are recognized and served as elephant flows, the proportionality in the allocation of network resources in relation with application semantics and application state will be far from optimal. A primary objective of this work is to address this gap and through that drive the value of software-defined architectures to data-intensive distributed applications.

III. Pythia Architecture

At the highest level of abstraction, Pythia is a distributed system software with two primary cooperating components, corresponding to the sensor/actuator paradigm: a) a Hadoop instrumentation middleware that runs on every Hadoop "slave" server (or Virtual Machine) and whose role is to predict future shuffle transfers at the level of mapper/reducer server (or Virtual Machine) pair during MR runtime and b) an orchestration entity that ingests - on a per job basis - future shuffle communication intent events and optimizes the network during runtime, aiming at reducing total job completion time. In the rest of this paper, we assume a bare-metal Hadoop deployment for the sake of space and thus use the term "slave server" to refer to the operating-system

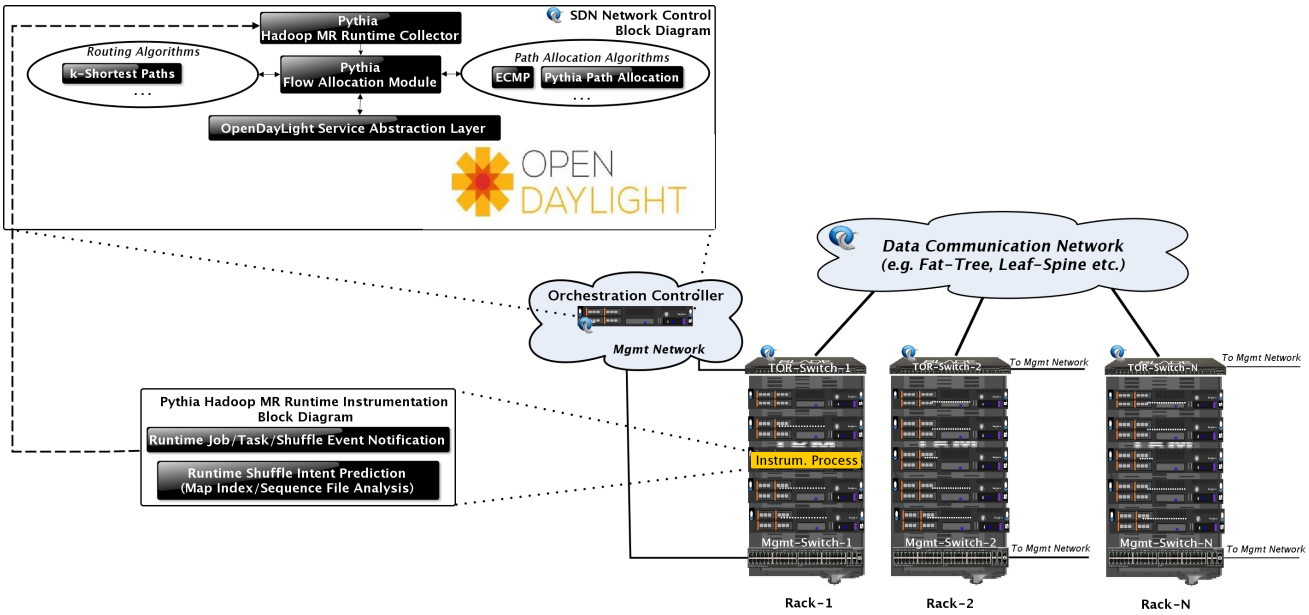


Figure 2: Pythia Architecture (right-hand side) and Control Software Block Diagram (left-hand side)

level entity that hosts a distinct Hadoop tasktracker. Still, it is important to note that our solution is fully compatible with Hadoop deployments in virtualized cloud environments too.

The right-hand side of Figure 2 depicts the architecture of the system within a reference cluster/datacenter infrastructure, comprising a set of server racks. The Hadoop cluster is typically deployed in a subset of this server pool. Intra-rack data communication (e.g., shuffling or HDFS block movement) occurs via one (or more) edge switches (Top of Rack - ToR switches) that all in-rack servers connect to, whereas the data communication network provides for inter-rack data communication. Pythia leverages the programmability offered by software-defined networks (SDN) to achieve fine-grained, timely and efficient allocation of network resources to shuffle transfers. Currently, Pythia supports a datacenter network compatible with the standard protocol realization of the SDN concept, namely OpenFlow [11].
Figure 2 shows also an additional management network that is physically distinct and typically of much lower bisection (and cost) to the data network, interconnecting all devices (servers, switches). This network is typically used for management/administration/provisioning purposes, for out-of-band control-/management-plane communication between OpenFlow switches and the network controller and - although not a prerequisite due to low network overhead incurred by our system (cf. Section V-C) - is also the physical network used to carry all control/monitoring traffic incurred by Pythia to minimize disruption to application data traffic.

At startup time Pythia initiates an instrumentation process at every server hosting a Hadoop tasktracker. As conveyed by the respective block diagram on the left-hand side of Figure 2, the instrumentation middleware constantly monitors its local tasktracker for job/task progress activity, while also providing for mapping of mapper/reducer identification from Hadoop namespace to network location (i.e., resolution of IP address per map/reduce task ID). Since Hadoop normally starts to schedule reducers only after a few mappers have been completed (by default 5%), it is expected that some flow intention detections will have unknown destinations. To remedy this, a collector's thread monitors for reducer initialization events and fills these incomplete shuffle intention entries with reducer destination information, as soon as the latter becomes available.
More importantly, each instrumentation process tracks its local tasktracker for newly spawned map tasks. At the event of a new map task creation, the instrumentation process locates the local file system path, where intermediate map task output will be spilled to, and subscribes to the local file system service for receiving asynchronous notifications, whenever new files are created under this path. Per Hadoop workings [12], intermediate output files are written to disk at map task completion time. Whenever the notification of such an incident is received by the instrumentation process (i.e., after a mapper has finished), it decodes the file(s) containing the intermediate map output and calculates the size of <key,value> pairs that correspond (and will be shuffled) to each one of the job's reducers. Last, it serializes the per-reducer predicted shuffle size in a message, together

with the ID of the respective map task, and transmits it to the Pythia collector server entity. It is important to note that the Pythia prediction instrumentation middleware is *transparent*, both to the Hadoop implementation, as well as to applications running on top, and thus can be seamlessly deployed on any existing Hadoop cluster.

The optimization and network programming part of Pythia is shown on the top, left-hand side block diagram of Figure 2, logically comprising a prediction notification collector, multi-path flow routing algorithm logic and a targeted flow allocation (or scheduling) block. As it will be extensively elaborated on in the next section, all these modules work in tandem to respond to shuffle prediction notifications and optimize flow scheduling in a manner that leads to faster shuffle phase completion time and thus to job acceleration. Currently, all of the SDN network control logic of Pythia is implemented in the form of modular components within an alliance OpenFlow controller project, namely OpenDaylight [13].

IV. Network Scheduling

In this section we describe the functionality and implementation of the Pythia network scheduling module. Essentially, the module ingests information about the physical network topology, the current link-level network utilization and the application communication intention and computes an optimized allocation of flows to network paths, such that shuffle transfer times are reduced; as a last step, it maps the logical flow allocation to the physical topology and installs the proper sequence of forwarding rules on the switches comprising the data network. The network scheduling module is implemented as a plugin module within OpenDaylight, a community-led, industry-supported open source OpenFlow controller framework. Internally, the network scheduling module consists of a Hadoop MapReduce runtime collector and a flow allocation module.

The Hadoop Runtime collector is responsible for receiving and aggregating the application-level information collected by each server's monitor (cf. Section III). In addition, the collector aggregates all flows that emanate from a distinct server (mapper) and are terminated to a distinct (reducer) server into a single flow entry that sums up the flow sizes of its constituents flows. Ideally, one would prefer to use the classical five-tuple definition of an application flow (<source-address,destination-address,source-port,destination-port,protocol-type>) to create OpenFlow forwarding entries during network programming. However, a Hadoop shuffle flow's TCP destination port number cannot be determined in advance (during prediction time), since it is only assigned by the sourcing server as soon as the flow starts (i.e., socket bind). Therefore, flow aggregation proves necessary. The hypothetical limitation of this is that it cancels the ability to apply differentiated network scheduling for reducer tasks running on the same server. In practice and throughout our experimentation, we have not identified the criticality of supporting such a feature as a performance booster. On the positive side, having a module supporting flow aggregation adds future flexibility to the Pythia system, particularly with regard to forwarding state conservation in softwared-defined networks (SDN). Given the high cost and thus limited size of the memory part of network devices storing so called "wildcard" rules (as is the case with four- or five-tuple rules) [14], large-scale future SDN network setups may force routing at the level of server aggregations (e.g., racks or sets of racks-PODs). Pythia can easily respond to such a requirement by populating the flow aggregation module with server location-awareness and an appropriate aggregation policy that maps flows to rack- or POD-pairs.

The flow allocation module implements both routing and flow allocation algorithms. During startup, it ingests topology events from OpenDaylight and generates a routing graph that represents the underlying multi-path network topology. Also during initialization, it computes the k-shortest paths among all server pairs in the network graph, where leaf vertices, intermediate vertices and edges represent servers, network switches and network links, respectively. The k-shortest path implementation uses hop-count as the distance metric. This module relies on the OpenDaylight topology update service to recompute the routing graph only when a change occurs in the physical network topology. By doing so and given that the k-shortest-path implementation that uses successive calls to the Dijkstra shortest-path algorithm is O(N^3) for small k, we are able to keep the routing computation overhead off the data path. Moreover, it provides fault tolerance, since the routing graph is updated at the event of link or switch failure.

The problem of optimally distributing flows among the available paths in a multi-path network to satisfy traffic demands is normally referred to as Multi-Commodity Flow problem, which is known to be NP-complete for integer/unsplittable flows [15]. However, there are a number of practical heuristics for simultaneous flow routing that can be applied to typical datacenter network topologies. In this work, we used a first-fit bin-packing heuristic to jointly allocate sets of predicted shuffle transfer flows to available paths. Our heuristic combines the link utilization information provided by the OpenDaylight link load update service with the communication intention information collected by our Pythia monitor to distribute the flows among the available paths. It also employs the knowledge of the application-level transfers to differentiate the portion of the network load that is due to shuffle transfers from background traffic (due to over-subscription). As such, it is possible to determine the amount of available bandwidth along a given path. Finally, the aggregated flows are assigned to the path that has the highest available bandwidth. We note that our design is modular enough to support further flow scheduling algorithms; the latter forms also part of our on-going work in this space.

The Pythia flow module handles only flows that are part of communication prediction for applications subscribed to the Pythia collector module. The rest of the datacenter traffic is handled through default datacenter network control processes. In this paper, we assume that flows - other than those handled by Pythia - are allocated to the available k-shortest paths via an ECMP-like (Equal-Cost Multi-Path) scheme. According to ECMP, all packets belonging to a distinct flow are hashed to the same output port (and thus path) at every intermediate network device, thus resembling a random, load-unaware flow allocation scheme. Our current ECMP implementation uses the five-tuple (<source-address,destination-address,source-port,destination-port,protocol-type>) to compute a flow hash and assigns a path to a flow based on a modulus computation on the flow hash value and the number of available paths in the routing graph.

V. Evaluation Results

A. Experimental Setup

Our experimental setup consists of 10 identical servers, each equipped with 12 x86_64 cores, 128 GB of RAM and a single SATA disk. The servers are organized in two racks (5 servers per rack) interconnected by two OpenFlow enabled Top-of-Rack (ToR) switches (IBM G8264 RackSwitch) with two links between them. A distinct server runs an instance of the OpenDaylight network controller and is directly connected to each of the ToR switches through a management network (as described in Section III). In terms of software, all servers run Hadoop 1.1.2 installed on top of Red Hat Enterprise Linux 6.2 operating system.

Since our setup has only a single HDD disk per server (with measured serial read rate of 130MBytes/sec) and multiple cores accessing it in parallel, we decided to configure Hadoop to store its intermediate data in memory. Otherwise, Hadoop would operate in a host-bound range (i.e., disk I/O rate would be the bottleneck), thus resulting in the setup being indifferent to any improvement brought to the network by Pythia. Having a balance between CPU, memory, I/O and network throughput is common in production-grade Hadoop clusters and therefore following the above practice is justified in the absence of Hadoop servers with arrays of multiple disks.

B. Job Speedup

As described in Section II, a reducer task does not start its processing phase until all data produced by the entire set of map tasks have been successfully fetched. Therefore, the shuffle phase represents an implicit barrier that depends directly on the performance of individual flows. Thus, even a single flow being forwarded through a congested path during the shuffle phase may delay the overall job completion time. In order to demonstrate the ability of Pythia to choose good paths to shuffle transfer flows and accelerate MapReduce applications, we conducted a number of experiments evaluating job completion times under different network over-subscription ratios. We used ECMP as baseline, since it has been used as the *de facto* flow allocation algorithm in multi-path datacenter networks. The various over-subscription ratios we experimented with are simulated by populating the network links with background traffic, specifically using the iperf tool to generate constant bit rate UDP streams.

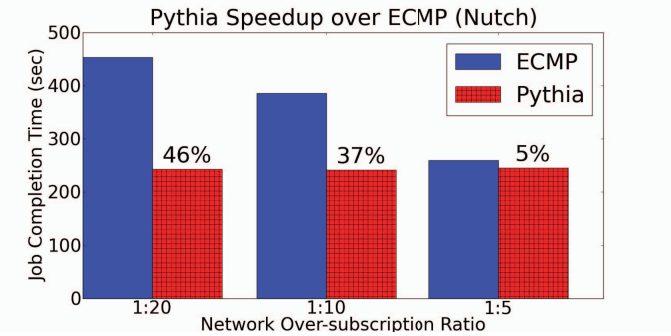


Figure 3: Nutch job completion times using Pythia (resp. ECMP) and relative speedup

We chose two benchmarks from the HiBench benchmark suite [6] that are known to be network-intensive: sort and Nutch indexing. The sort application is one of the examples that are provided by the Hadoop distribution. It is widely used as baseline to Hadoop performance evaluations and is representative of a large subset of real-world MapReduce applications (e.g., data transformation). We configured sort to use an input data size of 240GB. The Nutch indexing application is part of Apache Nutch [16], a popular open source web crawling/indexing software project, and is representative of one of the most significant uses of MapReduce (large-scale search indexing systems). We configured Nutch to index 5M pages, amounting to a total input data size of $\approx$ 8GB.

Figure 3 depicts Nutch job completion times using Pythia and ECMP respectively and the relative speedup. Times are reported in seconds and represent the average of multiple executions. As can be observed, Pythia outperforms ECMP for different over-subscriptions ratios. The maximum speedup was obtained for the 1:20 over-subscription ratio case, where Pythia improved job performance by 46%. It is worth noting that job completion times for Nutch using Pythia do not significantly increase by handing more network capacity to Hadoop and are comparable to the respective job completion time measured in a network without over-subscription (242 seconds in our setup). This indicates that the Pythia flow allocation algorithm, coupled with early flow size knowledge, manages to almost optimally assign the maximum capacity that Hadoop MapReduce needs.

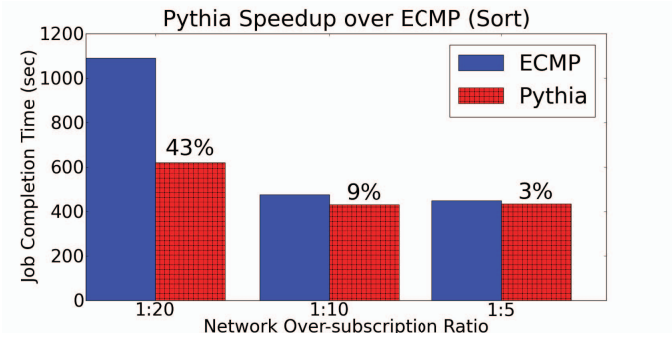


Figure 4: Sort job completion times using Pythia (resp. ECMP) and relative speedup

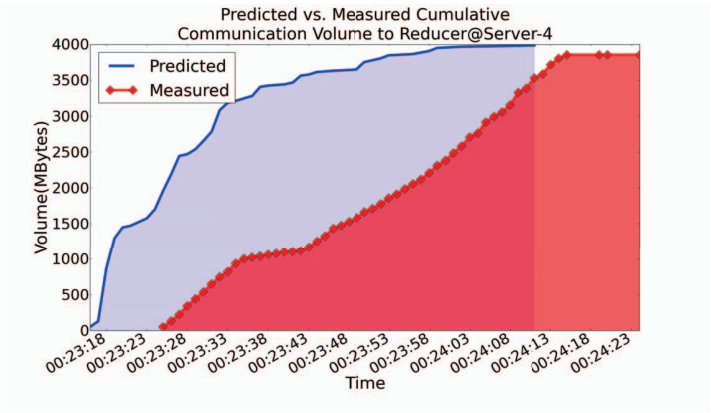


Figure 5: Prediction promptness/accuracy over time for traffic emanating from a single Hadoop tasktracker server (60GB integer sort job)

Figure 4 shows the results for the sort application. Unlike Nutch, sort jobs running over Pythia are not able to maintain similar job completion times over different over-subscriptions ratios. We believe this is due to the individual shuffle flow characteristics, particularly because the smaller flows created by Nutch increase the opportunity for optimization. However, Pythia is still able to outperform ECMP for different over-subscription ratios with a improvement of up to 43%.

C. Prediction Efficacy and Overhead

Given the value of the communication intent prediction middleware as a standalone component that could also be used in multiple other runtime optimizations of the Hadoop infrastructure beyond network scheduling (e.g., storage or early skew prediction), we elaborate here on the timeliness and flow size accuracy of the prediction middleware. To evaluate these metrics for the prediction middleware, we deployed NetFlow network monitoring probes across all servers comprising our testbed prototype, together with a NetFlow collector at a server that was connected to all servers comprising the Pythia prototype. Clocks on all servers in this setup were synchronized to 100ms accuracy. We then run multiple runs using various benchmarks, capturing both a) cumulative traffic volume over time that each Hadoop server sourced towards other Hadoop server nodes, as predicted by Pythia and b) all shuffle flow traffic (port 50060) exchanged between pairs of Hadoop servers in our setup using the NetFlow monitoring system. In addition, we post-processed the NetFlow traces to obtain cumulative, per-server sourced shuffle flow volume, compatible with the measurements we obtained from the Pythia predictor.

Figure 5 plots the outcome of this analysis for a single server sourcing shuffle traffic to the various reducers (Server-4). We observe that there is a substantial distance between the predicted and the measured traffic curves (approximately 9sec at minimum), which effectively translates to Pythia being able to predict the traffic volume that will exit a Hadoop server well in advance of the time that the actual traffic will start entering the network. This finding was consistent across all servers and throughout our experimentation with other workloads. Generally, the timeliness of Pythia prediction was found to be operating in a safe margin, relative to the time budget that contemporary networking hardware allows for programming the network at runtime (typically in the order of 3-5ms/flow installed). Intuitively, the timeliness of prediction depends on the time gap between a map task finish event and the event of a reducer task starting to fetch data from the finished mapper. Given that Hadoop limits the number of parallel transfers that each reducer can initiate at every instance of time (especially in larger-scale setups), we expect the above time gap affecting prediction timeliness not to be sensitive to Hadoop configuration parameter setup. We are currently working on modeling the problem using relevant Hadoop parameters as input and designing experiments to confirm this insensitivity, as part of our on-going work.

Pertaining to accuracy in predicting traffic volume, Pythia was always able to never lag the actual traffic measurement trace in terms of cumulative traffic volume sourced. As seen in Figure 5, Pythia is over-estimating traffic volume by a factor of 3%-7% for a single server. While we don't expect this churn to be detrimental or lead to measurable over-commitment of network resources, we believe that it has its source in the accuracy of how Pythia computes the protocol overhead that it adds to the shuffle flow prediction volumes collected from Hadoop servers (the instrumentation process works at the application-layer and therefore the protocol overhead that needs to be added to predict the flow volume as it will be seen "on the wire" is computed based on known protocol header sizes).

Last, we report on the overhead induced by the instrumentation middleware. Based on preliminary measurements, per Hadoop server average CPU and I/O overhead ranged from 2% to 5%, while memory occupancy overhead was

insignificant given our (commodity) server configuration. Intuitively, overhead comprises a constant ("dc") factor stemming from continuous monitoring of MapReduce task progress and a spike factor stemming from index file analysis at the event of a map task finish. Recognizing that the instrumentation overhead characterization merits further study, we plan to address it in our follow-up work, together with characterizing the network overhead incurred by Pythia prediction notification messages.

VI. Related Work

Due to the inherent nature of data-intensive distributed analytics frameworks to move large volumes of data between application server nodes, recent research work in the area has started focusing on optimizing against network bottlenecks (e.g., TCP Incast [17]) that may impede optimal performance of such workloads. Camdoop [18] manages to reduce the volume of data shuffled in MapReduce jobs by employing in-network combiners. Similarly, Yu et al. [19] move parts of the merge phase into the network, thus de-serializing the map from the reduce phase and bringing substantial improvement due to increased parallelism in phase execution. Both approaches can act complementary to our work and - even more - benefit from the advance knowledge of future shuffle flows that Pythia provides for.

Big data applications are among the obvious "beneficiaries" of the fine degree of programmability that the software-defined infrastructure movement brings along. Focusing on the data-movement part, Wang et al. [20] identify various opportunities for runtime network optimization to accelerate MapReduce jobs, potentially using an appropriate abstraction framework to interface applications with the infrastructure, such as Coflow [21] or MROrchestrator [22]. Orchestra [4] is a versatile framework showcasing the value of network-awareness (e.g., network-state aware scheduling of sets of interdependent flows) and/or network-optimized execution (e.g., by dynamically manipulating flow rates) to big data movement patterns (shuffle, broadcast). However, Orchestra requires explicit support from the big data framework it optimizes (e.g., Hadoop) and thus - unlike Pythia - cannot be used without reworking the design and implementation of the application framework. Still, should Hadoop reach a level that it interfaces with dynamic infrastructure orchestration frameworks like Orchestra, the integration of our system as a sub-component of such frameworks is rather straightforward.

Among all related work in the field, FlowComb [23] is a framework with significant overlap with our system: it employs shuffle-phase communication intention to apply intelligent, ahead-of-flow-occurence network optimization towards MapReduce performance improvement. In fact, the first public communication on FlowComb occured while we were already in the process of developing our prototype. Next though to the similarities, there are also subtle differences: network optimization (flow scheduling) in FlowComb does not leverage application intelligence (except from predicted flow volumes), even if the use-case driving the work grants access to such information. Instead, Pythia draws from past manifestations [4] about the value in taking flow criticality into consideration, therefore incorporating flow priority as a criterion in network optimization, in addition to flow sizes and network topology/state. At the engineering level, our implementation of predicting flows based on deep Hadoop index/sequence file analysis results in more timely prediction compared to the results communicated by FlowComb. Last, while recognizing that [23] reports on on-going work, the testbed used for the evaluation of FlowComb used only a single network over-subscription ratio (1:10 for 1Gbps server NICs) and was likely to exhibit high-latency due to using software switches. In this setting, it is hard to assess how FlowComb will perform in higher-capacity, production grade datacenter networks. Nevertheless, there is great value in the FlowComb work and its recent appearance strengthens the argument for the timeliness and relevance of the research reported in this paper.

VII. Conclusions

This paper proposed Pythia, a system that improves the performance of Hadoop MapReduce jobs through runtime communication intent prediction and fine-grained control of the underlying datacenter network. After motivating the relevance and need for the work presented herein through analysis of exemplary executions of Hadoop jobs and treatment of the incurred flows thereof by the network, we outlined the architecture of our system and elaborated in the functionality and algorithms embedded into its constituent components.

A good portion of the work has been invested in developing a system prototype within a downsized datacenter located in our lab. We presented quantitative results of our Pythia prototype, obtained in a 2-rack testbed setup and using hardware OpenFlow switches and various MapReduce workloads as input. Our evaluation manifests that Pythia achieves significant acceleration of the Hadoop workloads under test (up to 46%), bringing an improvement that varies depending on network capacity available to Hadoop and the specificities of the workload. Given the value of our prediction middleware as a standalone component, we also presented evaluation results manifesting its superior ability to timely predict flows well in advance prior to their occurrence in the network, together with its ability to predict flow sizes fairly accurately.

The contribution of the present work to faster Big Data analytics and thus reduced time-to-insight through acceleration of the Hadoop analytics framework is profound. And while most of the system work on Hadoop has focused on improving other parts of the framework (e.g., job scheduling, partitioning) or the underlying infrastructure (e.g., compute

resource allocation), we show through this work that there is great potential and value in optimizing large-scale analytics runtimes against the underlying network. From a more elevated perspective, we rate the present work as a tangible value case supporting the realization of large-scale distributed computing as a programmable stack, in accordance with the software-defined argument.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005

[3] IBM, P. Zikopoulos, and C. Eaton, *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*, 1st ed. McGraw-Hill Osborne Media, 2011.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, p. 98, Oct. 2011.

[5] Apache Software Foundation. (2013) Hadoop. [Online]. Available: http://hadoop.apache.org

[6] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, 2010, pp. 41–51.

[7] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications," in *5th Open Cirrus Summit*, 2011.

[8] C. Hopps. (2000) Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, IETF. [Online]. Available: http://tools.ietf.org/html/rfc2992.html

[9] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*. New York, NY, USA: ACM, 2009, pp. 51–62.

[10] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: dynamic flow scheduling for data center networks," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10. Berkeley, CA, USA: USENIX Association, 2010, p. 1919. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855711.1855730

[11] (2009) OpenFlow switch specification version 1.0.0. [Online]. Available: http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf

[12] T. White, *Hadoop - The Definitive Guide (2. ed.)*. O'Reilly, 2011.

[13] Linux Foundation. (2013) Opendaylight collaborative project. [Online]. Available: http://www.opendaylight.org

[14] R. Trestian, G.-M. Muntean, and K. Katrinis, "Micetrap: Scalable traffic engineering of datacenter mice flows using openflow," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, 2013, pp. 904–907.

[15] S. Even, A. Itai, and A. Shamir, "On the Complexity of Time Table and Multi-Commodity Flow Problems," in *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1975, pp. 184–193.

[16] Apache Software Foundation. (2013) Apache nutch. [Online]. Available: http://nutch.apache.org

[17] Y. Chen, R. Griffit, D. Zats, and R. H. Katz, "Understanding tcp incast and its implications for big data workloads," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2012-40, Apr 2012. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-40.html

[18] P. Costa, A. Donnelly, A. Rowstron, and G. O'Shea, "Camdoop: exploiting in-network aggregation for big data applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012.

[19] W. Yu, Y. Wang, and X. Que, "Design and evaluation of network-levitated merge for hadoop acceleration," *Parallel and Distributed Systems, IEEE Transactions on*, vol. PP, no. 99, 2013.

[20] G. Wang, T. E. Ng, and A. Shaikh, "Programming your network at run-time for big data applications," in *Proceedings of the first workshop on Hot topics in software defined networks (HotSDN '12)*. ACM Press, 2012, p. 103. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2342441.2342462

[21] M. Chowdhury and I. Stoica, "Coflow: a networking abstraction for cluster applications," in *Eleventh ACM Workshop on Hot Topics in Networks (HotNets-XI)*. ACM Press, Oct. 2012, pp. 31–36.

[22] B. Sharma, R. Prabhakar, S.-H. Lim, M. T. Kandemir, and C. R. Das, "MROrchestrator: a fine-grained resource orchestration framework for MapReduce clusters," in *IEEE 5th International Conference on Cloud Computing (CLOUD)*. Honolulu, HI, US: IEEE, Jun. 2012, pp. 1–8.

[23] A. Das, C. Lumezanu, Y. Zhang, V. Singh, and G. Jiang, "Transparent and flexible network management for big data processing in the cloud," in *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '13)*. San Jose, CA, US: USENIX, Jun. 2013.