# PHURTI: APPLICATION AND NETWORK-AWARE FLOW SCHEDULING FOR MAPREDUCE

BY

XIAO CAI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

# ABSTRACT

Traffic for a typical MapReduce job in a datacenter consists of multiple network flows. Traditionally, network resources have been allocated to optimize network-level metrics such as flow completion time or throughput. Some recent schemes propose using application-aware scheduling which can reduce the average job completion time. However, most of them treat the core network as a black box with sufficient capacity. Even if only one network link in the core network becomes a bottleneck, it can hurt application performance.

We design and implement a centralized flow scheduling framework called Phurti with the goal of decreasing the completion time for Hadoop MapReduce jobs. Phurti communicates both with the Hadoop framework to retrieve job-level network traffic information and the OpenFlow-based switches to learn about network topology. Phurti implements a novel heuristic called Smallest Maximum Sequential-traffic First (SMSF) that uses collected application and network information to perform traffic scheduling for MapReduce jobs. Our evaluation with real Hadoop workloads shows that compared to application and network-agnostic scheduling strategies, Phurti improves job completion time for 95% of the jobs, decreases average job completion time by 20% and tail job completion time by 13%.

*To my family, friends, and colleagues for their love and support.*

# ACKNOWLEDGMENTS

I would like to specially thank my adviser Professor Roy H. Campbell for his generous support and patient guidance. I greatly appreciate Shayan Saeed for his collaboration and his effort is critical to make this research project successful. I am grateful to Professor Indranil Gupta for his insightful advices and suggestions incorporated into the thesis. I would also like to thank Professor Brighten Godfrey, Chi-Yao Hong for sharing early views and thoughts for the project. Finally I would like to thank SRG group members who have generously helped me including Cristina Abad and Mirko Montanari.

This thesis is based upon research work supported in part by the following grants: NSF CNS 1319527, NSF CCF 0964471, and AFOSR/AFRL FA8750-11-2-0084.

# TABLE OF CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1 Overview

The shuffling phase (intermediate data transfer) in Hadoop [1] can account for 33% of the running time of a MapReduce job on average [2]. The shuffling traffic for a job contains multiple flows between host pairs, and the reduce phase of the job cannot start until all flows have finished. In a shared cluster with multiple jobs running, a job flow might be throttled by traffic belonging to other jobs and can become a straggler. Flow-based scheduling policies [3, 4, 5] decrease the average completion time of flows but they can starve the large flows, thereby increasing the completion time of the job. Consequently, it is important to have application-awareness while scheduling network flows.

In modern datacenters, it is common for multiple MapReduce jobs to share cluster resources. While CPU and memory can be allocated efficiently, it is very hard to control network usage since it is a distributed resource. This means that in addition to application-awareness, it is desirable to have network-awareness during flow scheduling for better application performance. Current network-aware traffic scheduling schemes [6, 7, 8] are focused on improving network utilization instead of application performance.

While other application-aware traffic scheduling techniques [2, 9, 10] have been proposed, our goal is to use both application and network topology information for allocation of network resources. Our approach can work in conjunction with the approach by Alkaff *et.al.*[11]. They utilize the application and topology information for task placement and choosing the network route while we perform flow scheduling and bandwidth allocation along predetermined network routes.

We design a centralized scheduling framework called Phurti which provides APIs to dynamically collect the shuffling phase traffic information from

Hadoop jobs, as well as network topology and flow routing path information from the OpenFlow-based [12] Software Defined Network (SDN) switches. Phurti provides the option to suspend or throttle the traffic of any job at any time. Unlike a decentralized architecture [10], our approach does not require any change in the network switches, thus making deployment easier. The information and functionality provided by Phurti in turn can be used by any flow scheduling algorithm.

We also design and evaluate a new heuristic called Smallest Maximum Sequential-traffic First (SMSF) that uses the application and network information collected by the APIs to schedule the MapReduce traffic. Our algorithm can preempt the flows based on job priority, utilize the network maximally and protect against starvation. To our knowledge, this is the first framework of its kind that collects and uses both the application and network information for scheduling the traffic for MapReduce jobs. Our approach works well when a majority of jobs are small and the datacenter network is congested. Both of these are generally true in real Hadoop clusters. Facebook traces for Hadoop workloads [13] show that more than 70% are small jobs less than 1 MB in size while [14] shows that network congestion is one of the main reasons for poor job completion times in MapReduce framework.

We deployed and evaluated Phurti on a cluster of 6 machines interconnected by 2 switches. We evaluate it using both microbenchmarks and realistic workload generated by the SWIM [13] Facebook workload. Evaluation results show Phurti improves job completion time for 95% of the jobs, decreases average job completion time by 20% and tail job completion time by 13%.

## 1.2 Thesis Outline

Chapter 7 examines some of the related works in. We present the general design of Phurti in chapter 3 and describe specific implementation details in chapter 5. We present our evaluations and their results in chapter 6 and conclude with chapter 9.

# CHAPTER 2

# MOTIVATION

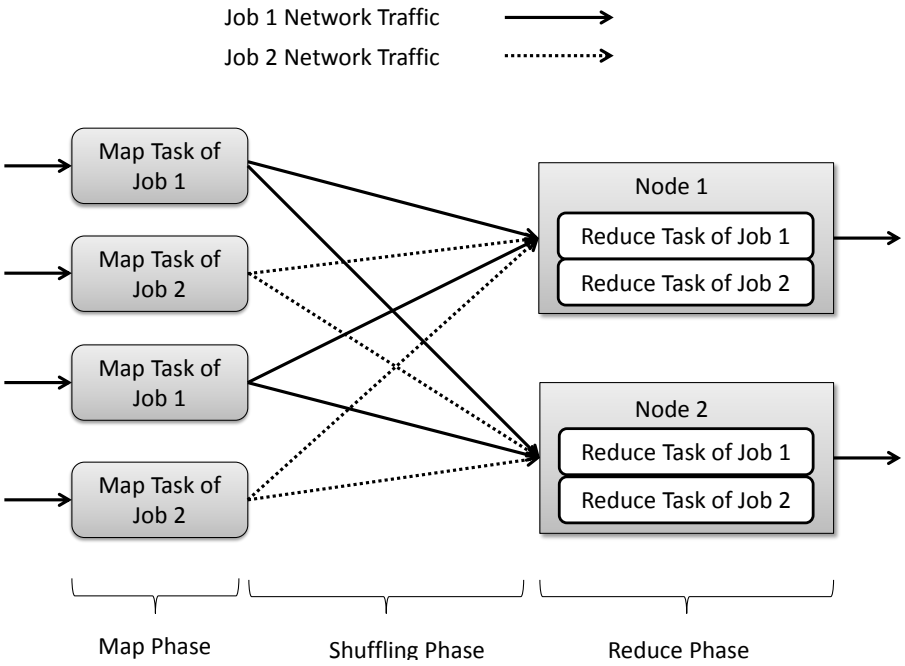



Figure 2.1: Traffic pattern for two Hadoop MapReduce jobs in a cluster.

## 2.1 Application-Awareness

The shuffling phase in a typical MapReduce job generates several flows in the network. A flow consists of all the traffic in a transport (e.g. TCP) connection. If two large MapReduce jobs happen to send data on shared network links simultaneously as shown in Fig. 2.1, they may slow each other down due to network contention. Since the computation for reduce function cannot start before all the flows in the shuffling phase complete, the overall job completion time depends on the successful completion of all of the constituent flows of that job.

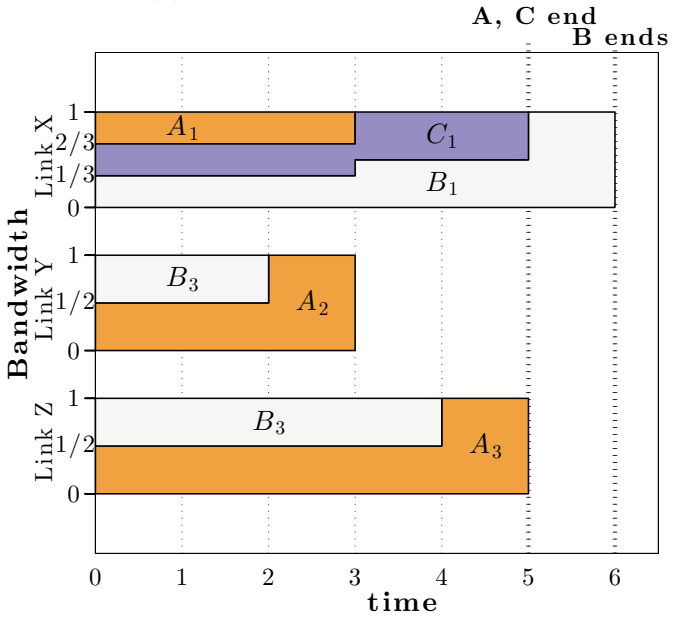
Flow-based scheduling policies such as shortest flow first (SFF) concerned

with optimizing flow level metrics such as flow completion time etc., have grown popular for datacenter networks. However, since flows of many simultaneous jobs are scheduled independently by such policies, they only perform well for network flow metrics and may not improve application performance. An *application-aware* scheduling strategy would take into account the workload characteristics and schedule all the flows of a job together. This would be more suitable for improving the average job completion time.

We demonstrate this using an example in Fig. 2.2. This shows three concurrent jobs A, B and C running on a shared cluster. A and B are larger jobs with three flows each, while C is a small job with only one flow. A fair sharing (FS) strategy such as DCTCP [3] (Fig. 2.2b) divides the bandwidth equally between the flows on shared links. For the example, all the jobs transmit concurrently on link X, so it becomes the bottleneck and increases the average job completion time to 5.33s. A flow based scheduling strategy, shortest-flow-first (SFF) [4][5] as shown in Fig. 2.2c, serializes the flows on each link and prioritizes the shorter ones on interfering links. This optimizes the average flow completion time. However, it schedules job A's flow first on link X and job B's flows first on link Y and Z. This leads to an increase in completion time for both jobs A and B because each job has straggler flows. We then show a simple application-aware scheduling strategy (Fig. 2.2d) that serializes the jobs and schedules all their flows together on different links. While this increases the average flow completion time from 3s to 3.43s, it improves the job completion time from 4.67s to 3.67s compared to SFF as shown in Fig. 2.2e. This behavior was also recognized by [9, 10].

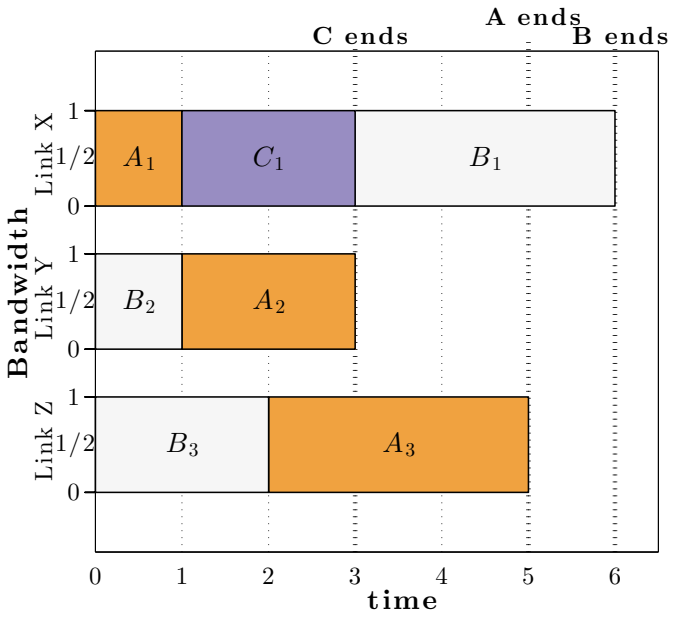
| ID | Link | Size |
|---|---|---|
| $A_1$ | X | 1 |
| $A_2$ | Y | 2 |
| $A_3$ | Z | 3 |
| $B_1$ | X | 3 |
| $B_2$ | Y | 1 |
| $B_3$ | Z | 2 |
| $C_1$ | X | 2 |

(a) Network Flows for jobs A,B,C

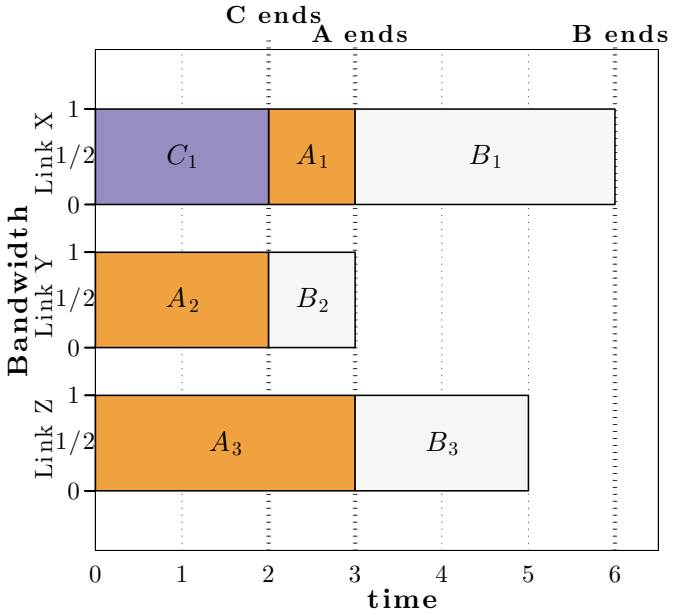



(b) Fair Sharing

Figure 2.2: Application-Aware vs. Application-Agnostic scheduling strategies for three concurrent jobs. Shortest Flow-first has the minimum average flow completion time (FCT) but an application-aware scheduler performs best in terms of average job completion time (JCT).

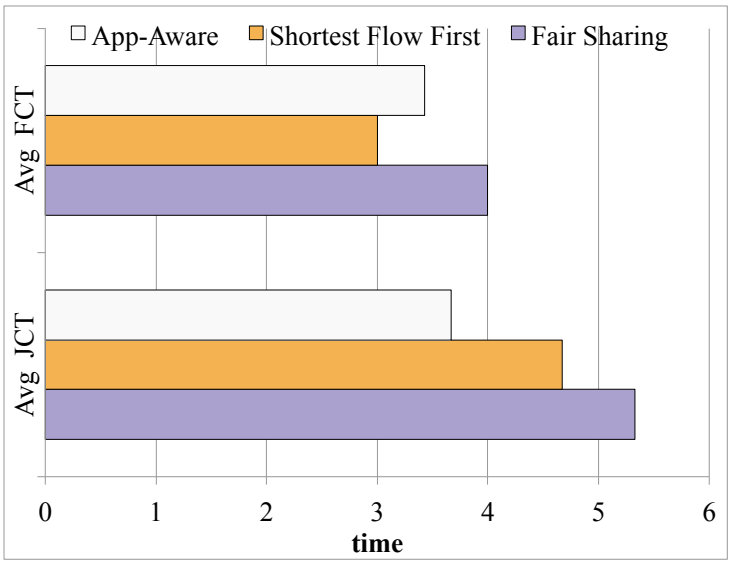



(c) Shortest Flow First




(d) Application-aware Scheduling

Figure 2.2: Application-Aware vs. Application-Agnostic scheduling strategies for three concurrent jobs. Shortest Flow-first has the minimum average flow completion time (FCT) but an application-aware scheduler performs best in terms of average job completion time (JCT).




(e) JCT and FCT for Scheduling Strategies

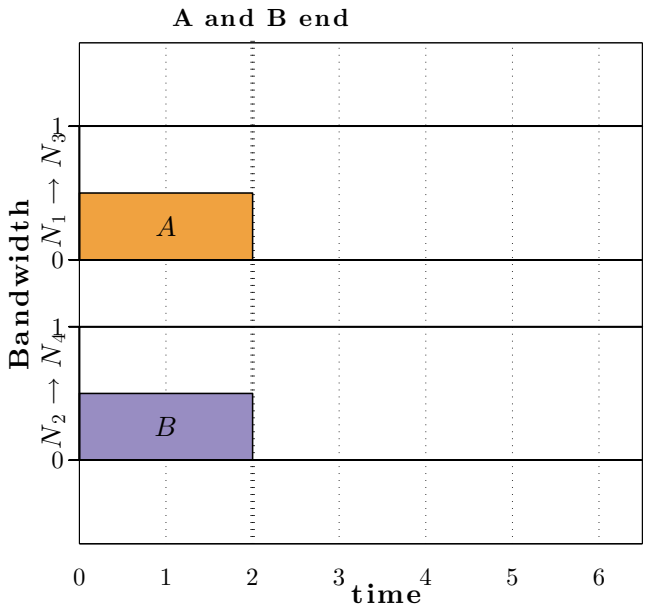
Figure 2.2: Application-Aware vs. Application-Agnostic scheduling strategies for three concurrent jobs. Shortest Flow-first has the minimum average flow completion time (FCT) but an application-aware scheduler performs best in terms of average job completion time (JCT).

## 2.2 Network-Awareness

Application-awareness alone is not sufficient for a scheduler to prevent concurrent jobs from slowing each other down. Even if the mappers and reducers of different jobs are scheduled on different nodes, their shuffling traffic might still interfere on a common link inside the network. A network-agnostic scheduler treats the network as a black box and assumes sufficient capacity at the core. It is unaware of any conflict between interfering flows, so it treats them as independent and schedules them concurrently. This can lead to a slowdown in data transfer if the link does not have sufficient capacity. A *network-aware* scheduler is aware of the network topology information. It can use this information to help prevent potential network congestion.

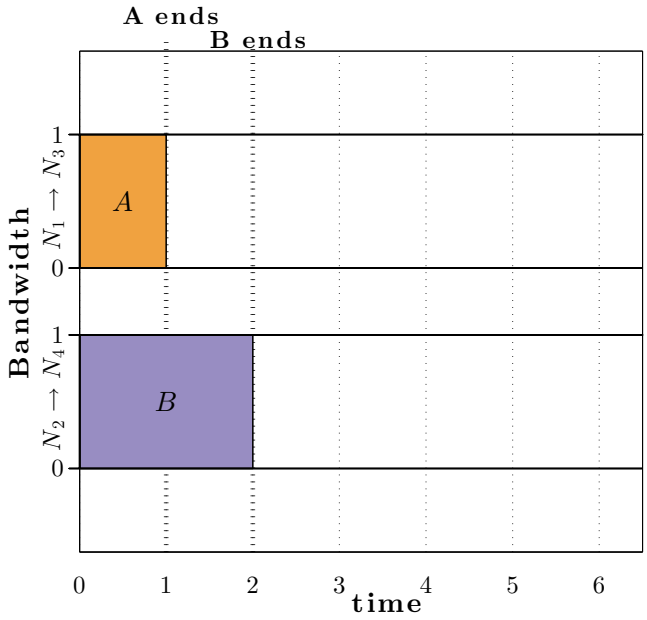
Fig. 2.3 considers two jobs each consisting of one flow of size 1. The topology of the network is shown in Fig. 3.1. All the links in the network have the same capacity. The flows do not share the end hosts but interfere in the network on the link between the switches S1 and S2. The network-agnostic scheduler (Fig. 2.3a) lets both the jobs send traffic at the same time. As

a result, they split the bandwidth of the bottleneck link S1→S2 between each other. This leads to a slowdown and both the jobs complete in 2s. A network-aware scheduler (Fig. 2.3a) would predict the flow *interference* and serialize the jobs. We define interference as the overlap of the paths of network flows from different jobs on at least one link. Initially, job A fully utilizes the link, completes and then job B can utilize the link fully. This reduces the average job completion time from 2s to 1.5s.




(a) Network-Agnostic Scheduling

Figure 2.3: Network-Aware vs. Network-Agnostic Scheduling for two concurrent jobs in the network in Fig. 3.1. Network awareness can reduce conflict in the network and improve the job completion time.




(b) Network-Aware Scheduling

Figure 2.3: Network-Aware vs. Network-Agnostic Scheduling for two concurrent jobs in the network in Fig. 3.1. Network awareness can reduce conflict in the network and improve the job completion time.

# CHAPTER 3

# SYSTEM ARCHITECTURE

To achieve application-aware and topology-aware network resource allocation, we design a flow scheduling framework called Phurti. The key idea of Phurti is to enable the applications and OpenFlow switches to pass the information about the system through APIs to enable global network traffic coordination. As shown in Figure 3.1, we propose a centralized architecture that communicates with the traffic-generating applications as well as with the OpenFlow switches. Phurti receives information about the underlying network topology, host placement and the path taken by each flow from the OpenFlow switches via its Southbound API. It also gathers information about the application generated network traffic by communicating with them via the Northbound API. We now discuss the design goals of these APIs. We will describe more implementation details in Section 5.

## 3.1 Northbound API

The Northbound API enables Hadoop to pass information about the shuffling phase traffic of each MapReduce job to Phurti. Whenever a MapReduce job launches, it contacts Phurti to register the job. It also sends notifications whenever it starts or stops sending the traffic into the network on a per-task basis. It can provide additional information to help in the scheduling decision, e.g., the size of network traffic a job needs to send between any pair of hosts during the shuffling phase, the number of concurrent flows in a job, etc. Phurti implements a rate-limiting module described in Section 5 that enables flow preemption. Depending on the flow scheduling algorithm, Phurti can choose to suspend, rate limit, or transfer a flow of any given job.

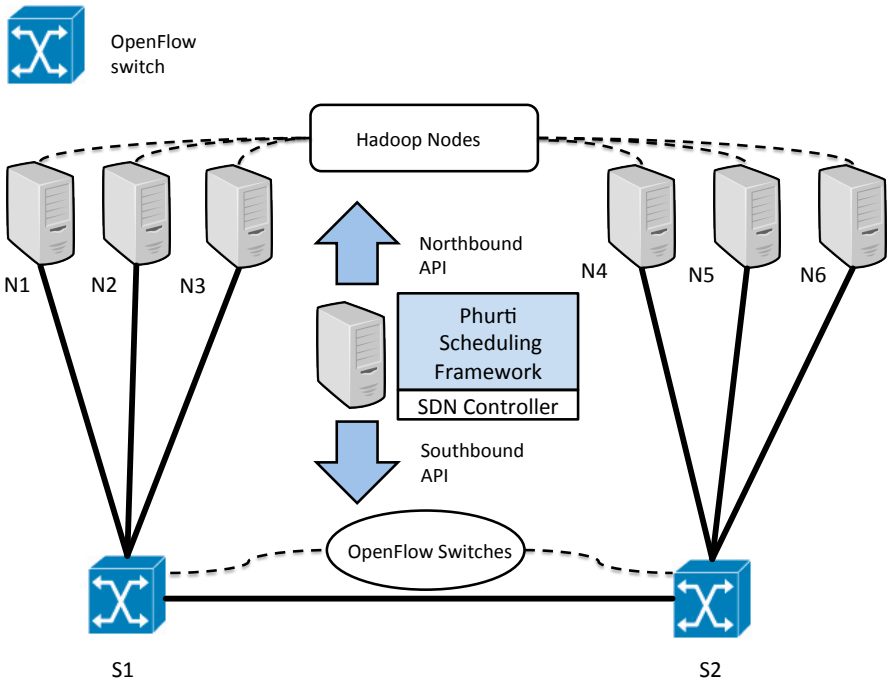



Figure 3.1: System Architecture of Phurti.

## 3.2 Southbound API

For topology-awareness, Phurti leverages the Southbound API to gain knowledge from OpenFlow switches about the cluster network topology including current hosts in the cluster network and how they are connected. It can also identify the complete path a flow traverses in the network. Acquiring this information allows Phurti to predict where interference of flows can happen to help avoid congestion.

# CHAPTER 4

# SCHEDULING ALGORITHM

In this section, we describe Phurti's scheduling heuristic with the goals of optimizing end-to-end job completion time and network utilization. We consider heuristics because scheduling the data transfers to minimize average completion time of shuffling is shown to be NP-hard, even without considering link-level capacity in core network [9].

## 4.1 Smallest Maximum Sequential-traffic First Heuristic

Shortest Job First (SJF) is a well known scheduling algorithm that can improve the average completion time of jobs. However, scheduling data transfers of MapReduce jobs purely based on the size of total amount of network traffic of the job can be inefficient. The completion time of the shuffling phase is likely determined by the size of largest amount of network transmission between any pair of hosts rather than the size of total amount of traffic. This is because the former is the bottleneck of the shuffling phase.

Concretely, we define the *sequential-traffic* $T_{ij}$ of a MapReduce job as the traffic it needs to transmit between host $i$ and host $j$. Note that sequential-traffic might consist of multiple flows. For a MapReduce job, we calculate the *Maximum Sequential-traffic* as $\max(T_{ij})$ across all host pairs $(i,j)$. In Fig 5.1, the maximum sequential-traffic of job J1 is 1GB, while for job J2 it is 2GB.

Using this, Phurti's flow scheduling strategy allocates network bandwidth to the flows of MapReduce jobs in increasing order of maximum sequential-traffic of jobs. We call this heuristic Smallest Maximum Sequential-traffic First (SMSF). We further discuss Phurti's mechanism for enforcing SMSF on flows in Section 4.2 and Phurti's bandwidth allocation strategy in Section

4.3.

Phurti maintains a priority queue for jobs. A job with smaller maximum sequential-traffic has higher priority. Phurti updates the priority queue continuously as it receives the new information from the Northbound API. When the size of maximum sequential-traffic of a MapReduce job changes, Phurti adjusts its priority accordingly.

## 4.2 Flows & States

In this section, we describe how Phurti enforces the priority defined by SMSF on network traffic generated by MapReduce jobs.

**Flow States:** A flow can belong to one of two possible states: TRANSMIT and SLOW. For a given link, only flows from one job can be in the TRANSMIT state. All the flows in the SLOW state share a small portion of the bandwidth of the links they traverse, while the majority of the bandwidth of links is used by the flows in the TRANSMIT state.

**Flow Entry:** When a flow arrives, Phurti retrieves the network path for the incoming flow. Phurti checks if there are any higher priority flows in the TRANSMIT state on any of the link along the network path of the incoming flow. If there are, the incoming flow is assigned a SLOW state, so that it does not interfere with the higher priority flows along its path. If not, it starts in the TRANSMIT state and asks Phurti to preempt other flows belonging to lower priority jobs along its path to the SLOW state. Phurti acquires all of these conflicting lower priority flows and rate-limits them to prevent interference. We summarize *Flow Entry* as pseudocode in Algorithm 1.

**Flow Exit:** Phurti keeps track of the states of all the flows and examines these states as flows finish. Firstly, if a flow finishes in the TRANSMIT state, Phurti retrieves all the links along its path and collects the flows in SLOW state that are traversing on those links. The collected SLOW flows are sorted in decreasing order of job priority. Each of the sorted SLOW flows is examined to check if it can be switched to TRANSMIT state, by the same procedure used in Section 4.2. Phurti preempts any flows if necessary.

Secondly, if the finished flow was in SLOW state, Phurti takes no action since no readjustment of bandwidth allocation is needed. *Flow Exit* is summarized as pseudocode in Algorithm 2.

**Algorithm 1** Flow Entry

```
1: function FlowEntry(flow)
2:     if canTransmit(flow, flow.path) then
3:         flow.state = TRANSMIT
4:         PreemptFlows(flow,flow.path,)
5:     else
6:         flow.state = SLOW
7:     end if
8: end function
9:
10: function canTransmit(flow,path)
11:     for Flow f along path do
12:         if flow.state == TRANSMIT AND flow.prio ≤ f.prio then
    ▷ interfere with a higher priority flow in TRANSMIT state
13:             return False
14:         end if
15:     end for
16:     return True
17: end function
18:
19: function PreemptFlows(flow,path)
20:     for Flow f along path do
21:         if f.prio < flow.prio AND f.state == TRANSMIT then      ▷
    interfere with a lower priority flow in TRANSMIT state
22:             f.State = SLOW
23:         end if
24:     end for
25: end function
```

**Algorithm 2** Flow Exit

```
1: function FlowExit(flow)
2:     if flow.state == TRANSMIT then
3:         S_FLOWS = {}
4:         for Flow f along flow.path do
5:             if f.state == SLOW then
6:                 S_FLOWS = S_FLOWS ∪ {f}
7:             end if
8:         end for
9:         for Flow sf ∈ S_FLOWS do
10:            if canTransmit(sf, sf.path) then
11:                PreemptFlows(sf,sf.path)
12:                sf.state = TRANSMIT
13:            end if
14:        end for
15:    end if
16: end function
```

## 4.3 Flow Managing Discussion

**Allowing Preemption:** Flows of a job may need to be preempted at any time due to the arrival of flows of higher priority jobs (with the priority defined by SMSF.) Without preemption, flows of low priority jobs can potentially hog network resources and increase average job completion time.

**Maximal Network Utilization** If there are two concurrent jobs in the cluster, our algorithm serializes them to let the higher priority job transfer first. However, some of the flows of the lower priority job might not interfere with the high priority job. If we use a strict policy to let only one job transfer at a time, the majority of network resources might be idle, which would be undesirable. This can decrease the network throughput compared to Fair Sharing, which would be utilizing the network maximally.

Phurti aims for a congestion-free maximal network utilization approach. Network flows of a MapReduce job can start with TRANSMIT state when it arrives, as long as it does not interfere with network flows from higher priority jobs (line 2 of Algorithm 1). These flows may be preempted anytime during their lifetime by the arrival of a higher priority job. This scheme ensures that the network is used fully by the incoming traffic.

**Starvation Protection:** If there is a continuous stream of high priority jobs arriving into the cluster, SMSF can lead to perpetual starvation for low

priority jobs. We present a two-fold solution to protect the jobs from getting starved as described below.

First, all SLOW flows with same source host $s$ share together a small fraction $\beta$ of $B$, where $B$ is the capacity of the link which connects $s$ with the core network. This approach is better than blocking the interfering flows of lower priority jobs. It allows the queued low priority jobs to make some progress, albeit small, even if they remain queued for a long time.

We also keep track of time elapsed since each job is submitted. Every $T$ seconds we check if a job has been submitted for more than *threshold* seconds and any of its flow is at SLOW state. For those flows, we switch them to TRANSMIT state for $\tau$ seconds. This ensures that all jobs can make steady progress towards completion without getting stuck behind short jobs perpetually. We mention the default values we use for these system parameters in Section 6.1.

# CHAPTER 5

# IMPLEMENTATION

In this section, we present implementation details about how Phurti interacts with Hadoop and OpenFlow switches.

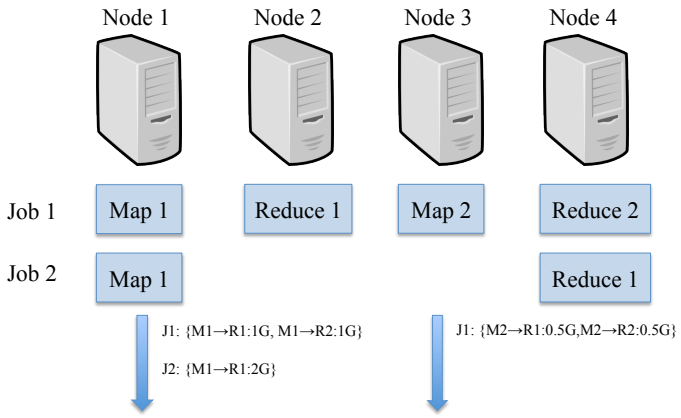
## 5.1 Northbound API

The Northbound API of Phurti provides a push-based notify function that can accept different types of messages from Hadoop for communicating the traffic information of different jobs.

**Job Registration and Unregistration:** When a MapReduce job starts, it registers itself by calling notify(JOB_START,jobID). Phurti adds it to the list of active jobs and initializes relevant states to get ready for future notifications. When a job finishes, it unregisters itself by calling notify(JOB_COMPLETE,jobID). Phurti cleans up the data structures allocated to keep track of the job.

**Task Host Notification:** When a map or reduce task launches, it calls notify(TASK_HOST,jobID,taskID,host) to notify Phurti of the host this task is running on.

**Partition Size Notification:** When a Map task completes, it notifies Phurti the size of intermediate data, by calling notify(SIZE,jobID,taskID, sizeInformation[]). sizeInformation contains information about the amount of data this map task needs to send to each reduce task.




| Path | Job-ID | Data Size |
|---|---|---|
| $N_1 \rightsquigarrow N_2$ | $J_1$ | 1G |
| $N_1 \rightsquigarrow N_4$ | $J_1$ | 1G |
| | $J_2$ | 2G |
| $N_3 \rightsquigarrow N_2$ | $J_1$ | 0.5G |
| $N_4 \rightsquigarrow N_2$ | $J_1$ | 0.5G |

Figure 5.1: Constructing Traffic Pattern.

In Fig. 5.1, we show an example of how Phurti uses the SIZE notifications. There are two MapReduce jobs J1 and J2. J1 has two map tasks: M1 at node 1 and M2 at node 3, and two reduce tasks: R1 at node 3 and R2 at node 4. J2 has one map task M1 at node 1 and one reduce task R1 at node 4. When the map tasks finish, M1 of J1 notifies Phurti it has generated 1GB data for each of the reduce tasks while M2 of J2 notifies it has generated 0.5GB data for each of the reduce tasks. Based on this traffic data and the host information of tasks learned through TASK_HOST notification, Phurti constructs the flows for all ongoing jobs.

**Flow Registration and Unregistration:** When the data transfer from a map task to a reduce task starts, Hadoop notifies Phurti of the source and destination as well as the size via notify(FLOW_REQUEST,jobID,flowID, flowInformation). Phurti keeps track of the state of ongoing flows including the network paths they traverse, in order to predict where flow interference can happen and make corresponding scheduling decisions. When the data transfer completes, Hadoop notifies Phurti by calling notify(FLOW_COMPLETE,jobID,flowID). This results in actions in Section 4.2.

## 5.2 Southbound API

Phurti uses the Southbound API to discover underlying network topology and predict flow interference.

**Path Retrieval:** We use the topology discovery and host tracker modules provided with the POX controller to obtain the network topology. Phurti uses the network topology information to query the path a network flow traverses. When Phurti needs to identify the path a flow traverses through the network, it calls query(GET_PATH,source,destination) where source and destination are the endpoints of the flow. The path returned consists of all links this flow traverses in the network.

Since SDN controller and OpenFlow switches together are continually tracking the network topology, even if there are changes in the network topology caused by adding or removing switches or hosts, Phurti will be capable of detecting these changes.

**Interference Avoidance**

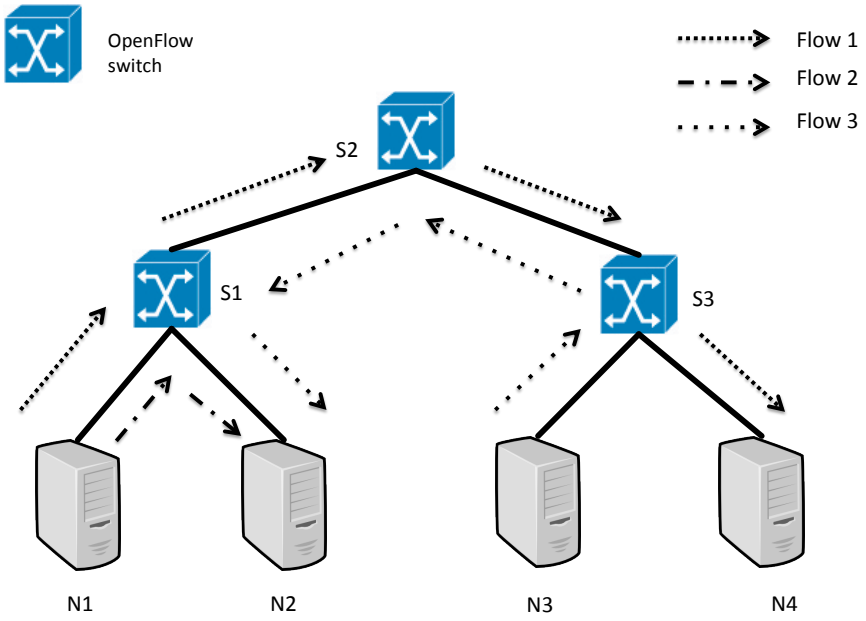



Figure 5.2: Predicting Flow Interference.

Phurti predicts possible flow interference by calling query(CHECK_PATH, PATH1,PATH2). It returns true if two flows intersect at any link in the network. Phurti queries the paths for the flows by using the GET_PATH query

and then uses the CHECK_PATH query to detect if those paths intersect on a certain link.

An example is shown in Fig. 5.2. Phurti uses GET_PATH query to retrieve the paths P1, P2 and P3 for flows Flow1, Flow2 and Flow3 respectively. query(CHECK_PATH,P1,P2)) returns True since both of them traverse on the same link (N1→S1). query(CHECK_PATH,P1,P3)) returns False since there is no overlap between P1 and P3, assuming all the switches are full-duplex. Phurti uses this information to predict flow interference and help avoid congestion.

# CHAPTER 6

# EVALUATION

## 6.1 Experimental Setup

We evaluate Phurti with a local testbed running Hadoop YARN 2.3.0. We evaluate it using both micro-benchmarks and a realistic workload based on production Hadoop trace from Facebook. We measure and compare average job completion time by Phurti with other existing approaches.

Our cluster consists of 6 servers (nodes) divided into 2 racks, where each rack consists of 2 nodes with 6 GB and 1 node with 3 GB RAM configured for Hadoop YARN containers. There are two HP 3500 OpenFlow switches each connected to 3 nodes of the same rack. Both the switches are connected by a singe link. The network topology is shown in Fig. 3.1. All the ports of the switches are capable of supporting 100 Megabits/sec full-duplex bandwidth. We use POX [25] as the OpenFlow controller. We run Phurti at a separate server with 2.40GHz CPU and 4 GB memory.

For the system parameters mentioned in Section 4.3, we set $\beta$ to be 1%, *threshold* to be 100 seconds, $T$ to be 20 seconds and $\tau$ to be 10 seconds. We found empirically these parameter values achieve balance between starvation protection for large jobs and avoid penalizing the short jobs for workloads we used.

## 6.2 Microbenchmarks

**Workload**: We use a workload of two MapReduce Terasort jobs to explicitly compare SMSF used by Phurti against other existing scheduling techniques. The first job has input size of 1GB and the second job has input size of 500MB. We adjust the inter-arrival time of the jobs so the 500MB job starts

its shuffling phase just after the 1GB job. We use job completion time as the primary metric. We compute average job completion time over 10 iterations.

**Phurti vs. Other Scheduling Techniques**: Fig. 6.1 compares Phurti with two other scheduling algorithms: Fair Sharing and JobFIFO. Fair Sharing (FS) is the default TCP fair sharing policy. Fair sharing allocates the link bandwidth equally between all the flows on that link. JobFIFO allocates links to the jobs in the order their network flows request them.

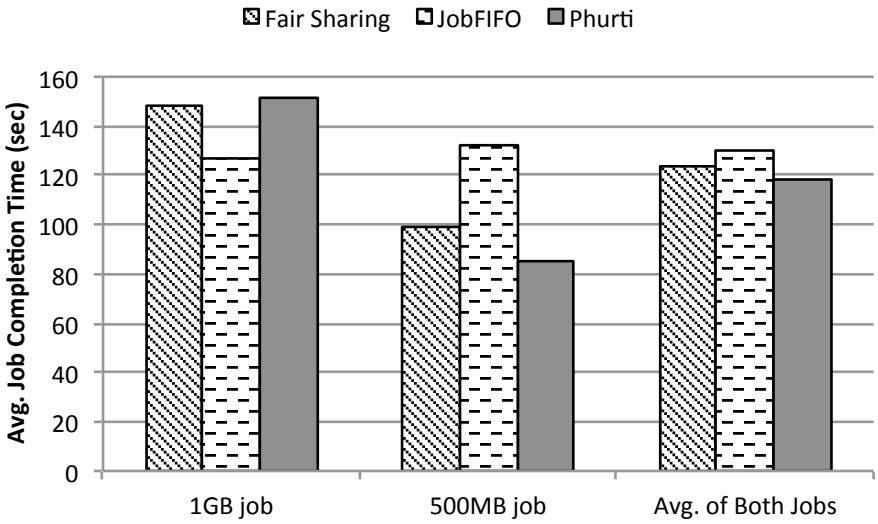
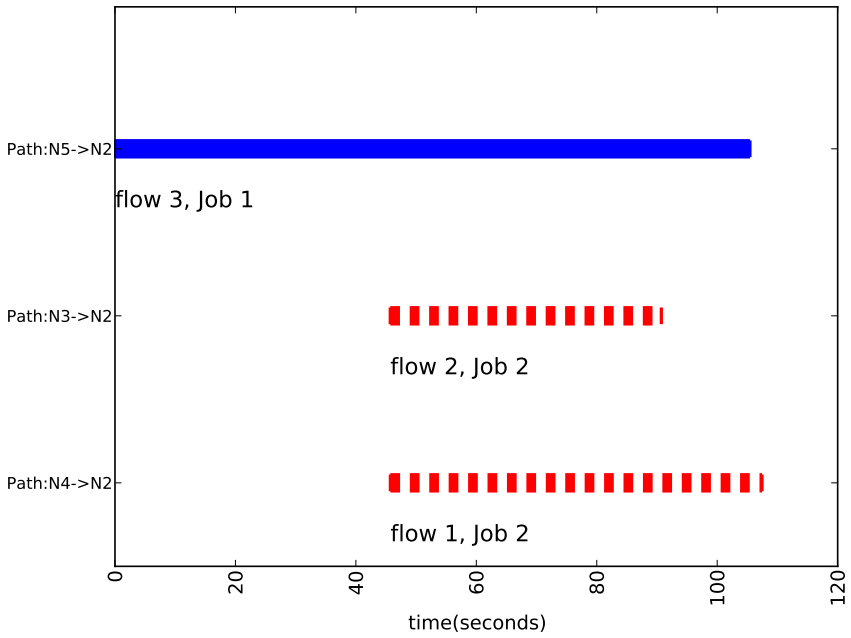



Figure 6.1: Comparison of SMSF used by Phurti with other scheduling algorithms for two Terasort jobs. SMSF has better average job completion time.

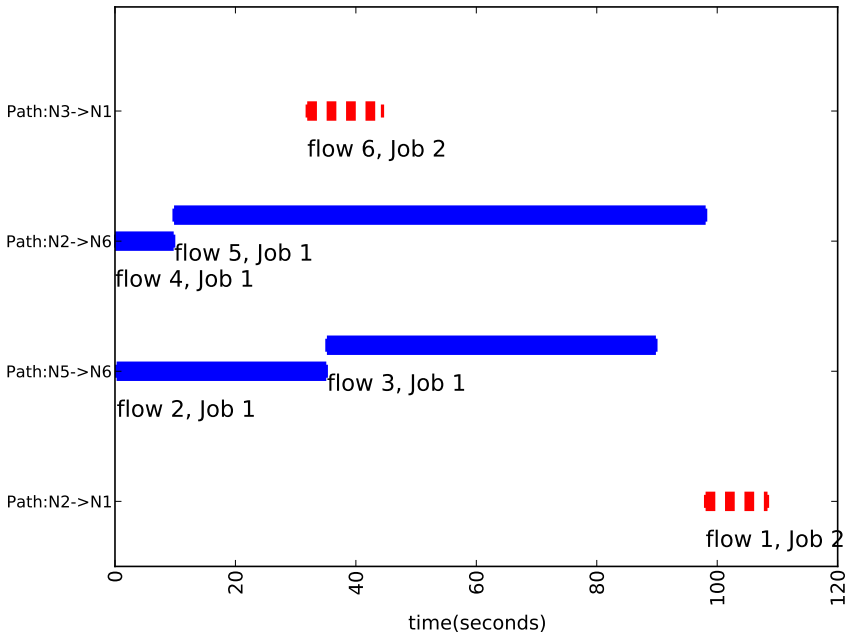
Fig. 6.1 shows that Phurti achieves the best overall average job completion time for the two jobs. The 1GB job has lowest average job completion when using JobFIFO because its network flows request the links before the 500MB job, and can transfer using full link capacities. The tradeoff is that JobFIFO has the worst average job completion for the 500MB job since it has to wait for the completion of the flows from 1GB job to start transmitting. Since FS allocates bandwidth equally for competing flows, it achieves a performance balance between 1GB and 500MB jobs compared to other scheduling policies as expected. We observe Phurti performs particularly well for the shorter job, in terms of reducing completion time by 36% compared to JobFIFO and 15% compared to FS. Furthermore, the penalty Phurti introduced for the completion time of the 1GB job is minimal (around 2.2% compared to FS) since Phurti is work-conserving: flow rates are re-adjusted as any flow completes to make sure no network link is unnecessarily under-utilized.

**Preemption and Maximal Network Utilization**: In order to further understand the performance of different flow scheduling techniques, for each of FS, JobFIFO and Phurti, we choose one of the ten iterations and visualize the timeline of flow transfers. This is shown in Fig. 6.2. For all of the plots, we use blue solid horizontal lines to show the flows of the larger 1GB job (job 1) and red dashed horizontal lines to show the flows of the smaller 500MB job (job 2). We label the paths traversed by each network flow on the left side of the plots.

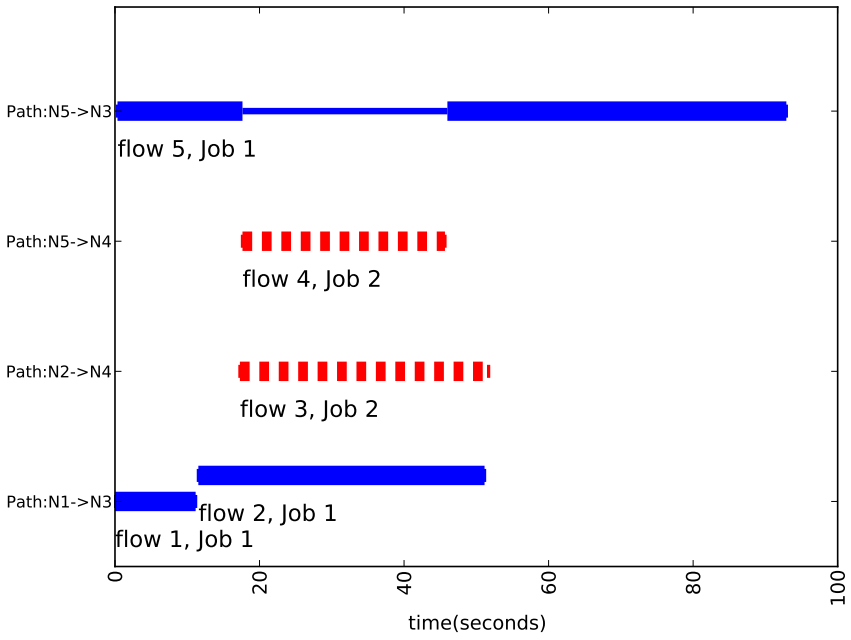



(a) Fair Sharing

Figure 6.2: Flow Timelines for Flow Scheduling Algorithms for two Terasort jobs. Solid lines show the flows for larger 1GB job while dashed lines show the flows for the shorter 500MB job. It shows preemption, rate limiting and maximal network utilization for Phurti.




(b) Job FIFO




(c) Phurti

Figure 6.2: Flow Timelines for Flow Scheduling Algorithms for two Terasort jobs. Solid lines show the flows for larger 1GB job while dashed lines show the flows for the shorter 500MB job. It shows preemption, rate limiting and maximal network utilization for Phurti.

We first start with FS shown in Fig. 6.2a. Both jobs have node N2 as the destination for their flows and as a consequence flows from different jobs encounter congestions at link $S1 \rightarrow N2$. Both jobs are only able to finish their shuffling phase after 100s.

Fig. 6.2b shows the flow transfers for JobFIFO. Since the large job (Job 1) starts its shuffling phase before the small job (Job 2) does, links on paths $N2 \rightsquigarrow N6$ and $N5 \rightsquigarrow N6$ are all allocated to flows of the large job. Flow 6 of the small job is still able to transmit concurrently with the flows of the large job, because flow 6 does not interfere with them. However, flow 1 of the short job has to wait for flow 5 of the large job to finish, because flow 1 interferes with flow 5 on link $N2 \rightarrow S1$. This increases the completion time of the small job that gets starved behind the large job.

Finally, we show Phurti's capabilities of preemption and achieving maximal network utilization in Fig. 6.2c. When the small job requests to transfer flow 3 and flow 4, flows of the large job are already transmitting. Flow 3 can start transmitting without interference with flows of the large job. Phurti predicts the interference between flow 4 and flow 5 on link $N5 \rightarrow S1$, and determines that the small job has a higher priority. Phurti preempts flow 5 (shown by reduced width) and starts transmitting flow 4, which allows the small job to finish its shuffling phase faster compared to the other cases. Right after flow 4 finishes transmission, Phurti lets flow 5 transmit at normal rate. This shows Phurti's work-conserving feature for achieving maximal network utilization, since none of the links are underutilized at any time.

## 6.3 Realistic Workload Evaluation

In order to demonstrate the advantages of Phurti for a realistic workload, we generate MapReduce jobs using SWIM[13], based on real MapReduce trace from Facebook cluster. The generated workload consists of 100 jobs. The original workload was collected on a 600-node cluster and we scaled down the jobs proportionally according to our testbed. The scaled trace still maintains original workload's characteristics including job arrival time, job size distribution and time variants of cluster utilization.

We divide the jobs in the workload into three categories, based on the size of intermediate data they generate: small, medium and large. Table

6.1 shows the percentage of the number of jobs belonging to each category along with the percentage of the intermediate data size. This shows that the majority of jobs in the workload are small jobs.

Table 6.1: Categories of Jobs in Workload.

| Job Size | % of Jobs | % of Bytes in Intermediate Data |
|---|---|---|
| Small | 62% | 5.5% |
| Medium | 16% | 10.3% |
| Large | 22% | 84.2% |

**Improvement of Job Completion Time**: We first show the benefit of Phurti via the main metric of job completion time. For each job in the workload, we take the difference between its job completion time under Phurti and its job completion time under FS to show performance improvement. A negative difference for a job shows its job completion time is smaller under Phurti. We average the results over 8 iterations.

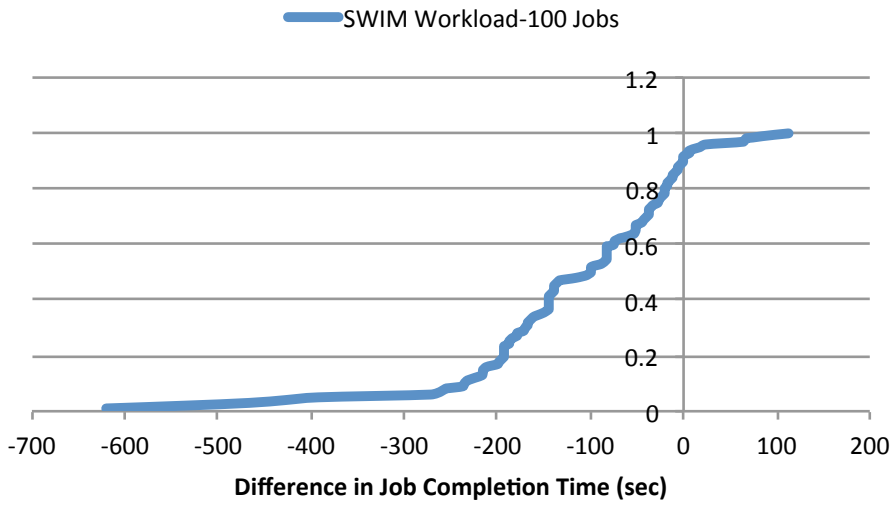



Figure 6.3: CDF of difference in Job Completion Time(sec): Phurti vs FS. Negative values imply Phurti is better.

In Fig. 6.3, we plot the CDF of the differences in job completion time for all jobs. The result shows around 95% of the jobs have improved job completion time under Phurti compared to FS. Around 50% of the jobs have at least 100s improvement in job completion time. Around 5% of the jobs have higher job completion time under Phurti and the worse increase in job completion time is around 100s. Compared to the fraction of jobs for which

the job completion time get improved and the factor of improvement they receive, we believe the trade-off introduced by Phurti is reasonable.

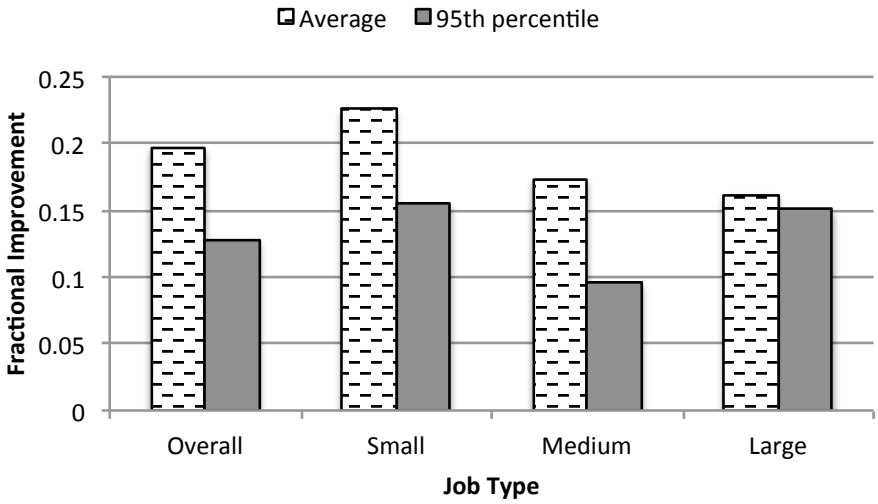



Figure 6.4: Fractional Improvement by Job Category. Phurti performs best for small jobs.

To further understand which of the jobs benefit the most under Phurti and how this improvement compares to FS, we plot Fig. 6.4. For each of job categories, small, medium and large, we compute both the average and 95th percentile of the job completion time. We show the results as a form of fractional improvement over FS. Among all jobs, Phurti achieves an average fractional improvement close to 20% and a 95th percentile improvement of nearly 13%. As expected, the small jobs have highest average fractional improvement of nearly 23% and 95th percentile improvement of 16% among all job categories. This is expected since jobs with smaller size are likely to have smaller maximal sequential-traffic, and thus have higher priority under Phurti. This is significant since there are much more smaller jobs in our MapReduce workload (62%) which is also confirmed by [13] (70%).

**Starvation Protection**: Although our workload is dominated by small jobs which have higher priority under Phurti, it should be pointed out that the large job category is still able to achieve an average fractional improvement of over 16% with 95th percentile improvement of 15%. This demonstrates that Phurti performs well for large jobs by avoiding perpetual starvation. This is also evident through the tail completion times. The 95th

percentile completion time for Phurti shows significant improvements greater than 10% over FS for all job categories.

**Impact on Network Utilization:**: Is Phurti able to achieve high network utilization? In order to answer this question, for each job we compute the *effective transmit rate* as fraction of time its flows spend in TRANSMIT state under Phurti. We plot the CDF of effective transmit rate in Fig. 6.5. Over 90% the jobs have effective transmit rate larger than 0.8, which means their flows spend more than 80% of the time in TRANSMIT state. The effective TRANSMIT rate for all the flows on average is greater than 0.9. Based on this result we conclude that, Phurti is able to maintain high utilization of the cluster network since flows transmit at their full potential most of the time (more than 90%).

In Fig. 6.6, we further analyze the network utilization of Phurti by showing the average effective transmit rate for jobs in different size categories. The results show that the small jobs have highest average effective transmit rate of nearly 95%, which is due to their higher priorities under Phurti. We observe that average effective transmit rate even for larger jobs is around 85%, despite the fact their flows have the lowest priorities and are likely to be preempted more frequently by other flows. This shows Phurti is able to achieve high network utilization for large jobs.

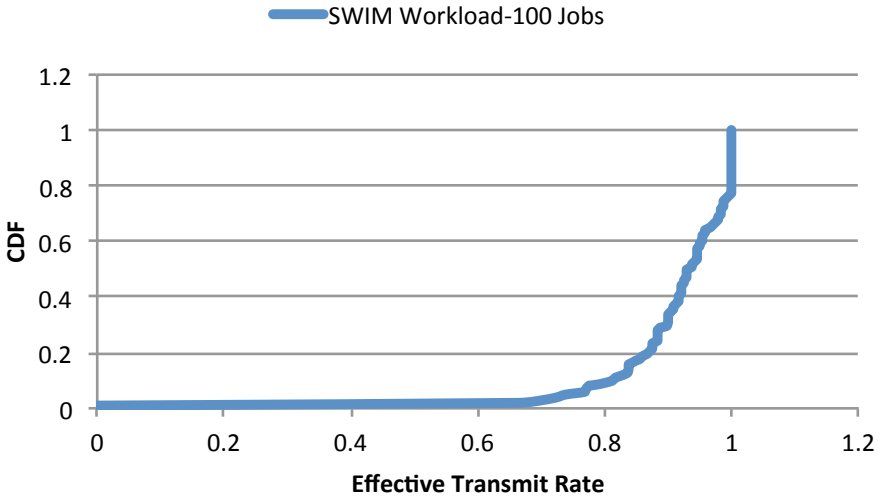



Figure 6.5: CDF of Effective Transmit Rate. Higher values are better.

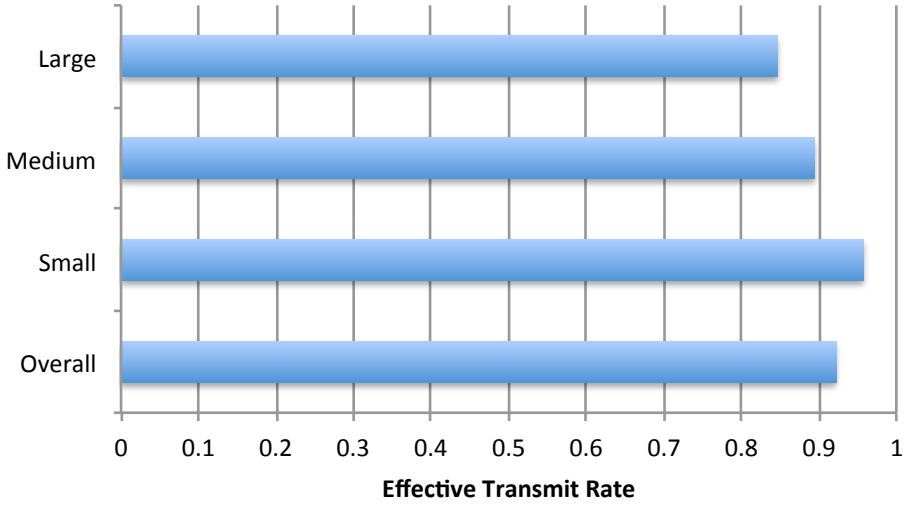



Figure 6.6: Average Effective Transmit Rate by Job Category. Small jobs have the best effective transmit rates.

# CHAPTER 7

# RELATED WORKS

**Traditional Flow Scheduling and Traffic Engineering:** There is rich literature about flow scheduling and traffic engineering techniques targeting only network-layer metrics. Both PDQ [5] and pFabric [4] can be used to approximate shortest-flow-first policy which is optimal for reducing average flow completion time but may lead to increased job completion times. Hedera [6] performs dynamic flow scheduling in a data center network to optimize network capacity but does not consider the application requirements during scheduling. SWAN [8] and B4 [7] are software defined WANs which use a centralized controller to perform traffic engineering to improve network utilization but are not concerned with application performance. Unlike Phurti, all of them are concerned with improving network level metrics but not application performance.

**Performance Optimization for Data-Parallel Computing:** Task-level optimization for data-parallel computing has been widely studied by research community. SUDO [15] is an optimization framework which analyzes user-defined functions to avoid unnecessary data-shuffling. RoPE [16] adapts execution plans based on estimates of user-defined code and data properties. Natjam [17] uses job-level and task-level eviction policies to enforce the job priority constraints for Hadoop jobs, but does not enforce network level scheduling. PACMan [18] is a distributed cache service analogous to Phurti but does cache management instead of flow scheduling. It prioritizes jobs with smaller wave-widths (number of parallel tasks). We believe Phurti can work along with SUDO, RoPE, Natjam and PACMan to improve the overall job completion time. Phurti does not interfere with the computation processing, memory and storage part of MapReduce jobs that can be efficiently scheduled and improved upon independently.

**Application-Aware and Network-Aware Task Schedulers** There are frameworks which use application and cluster information to allocate

resources to tasks. Tetris [19] is a scheduler that assigns tasks to machines based on their requirements for resources such as CPU, memory, storage and network with priorities based on smallest remaining time first. Wang *et al.* [20] propose using application and network-awareness in schedulers for scheduling jobs and do run-time network configurations to jointly optimize application performance and network utilization. Alkaff *et al.* [11] propose a cross-layer scheduler between the application and the networking layer. It uses the application and network information to perform task placement and select network routes. However, these schedulers work on allocating the nodes and network routes to tasks while Phurti performs flow scheduling on precomputed paths based on job priority.

**Application-Aware Traffic Scheduling:** Recent work has started to explore the opportunities to optimize application performance by implementing application-aware cloud network. Ferguson *et al.* [21] provide an API for SDN that allows the applications to formulate an overall network policy. FlowComb [22] uses software agents to predict application network transfers and avoids network congestion by scheduling upcoming flows via a centralized decision engine. Chanda *et al.* [23] describe a traffic engineering scheme which uses metadata such as content length to optimize the content delivery.

Literature that is most closely related to Phurti includes Orchestra [2], Baraat [10] and Varys [9]. Orchestra uses Weighted Shuffle Scheduling to minimize the completion time of a shuffle. However, Orchestra relies on launching multiple TCP connections to adjust flow transfer rate. Instead, Phurti uses explicit rate limiting mechanism, which adjusts flow transfer rate faster and incurs lower traffic overhead. Baraat utilizes a decentralized task aware scheduling system to minimize the task completion time. It assigns flow priorities in a task-aware fashion for scheduling. Phurti uses a centralized framework with a different scheduling strategy to schedule the network flows. Baraat's approach is at transport layer and requires modifications to both end-hosts and switches, while Phurti is transparent to client application and underlying network.

Coflow [24] proposes a networking abstraction for cluster applications to express their communication requirements. Varys uses this abstraction to implement an inter-coflow scheduling policy for improved and predictable communication time. While Phurti priorities job transfers in a similar fashion with Varys, Phurti differs with Varys in two important aspects: i) Phurti

is network topology-aware, ii) Phurti can schedule a subset of flows of a MapReduce job as soon as they are ready and thus can achieve high network utilization.

# CHAPTER 8

# DISCUSSION

## 8.1 System Design Trade-offs

Phurti contains the weaknesses of common centralized architectures including scalability and single point of failure. On the other hand, implementing a centralized scheduling scheme allows us to move the flow states away from switches. It also makes it easier to maintain consistency for flow scheduling decisions.

## 8.2 Task Placement via Network Layer Feedback

Phurti tries to schedule the network flows to accommodate the network resource demand of cluster applications. An orthogonal approach is to let Hadoop place the tasks on hosts based on network conditions. We believe that Phurti's traffic scheduling can be integrated with such task placement to further improve application performance.

## 8.3 Routing Decisions via Application Feedback

Currently we are using the Southbound API of Phurti to pull the network information only. However, because we use OpenFlow switches, the API also provides the option to perform routing decisions. We leave it for future work to come up with a scheme that exploits multiple paths for application-aware routing strategies.

# CHAPTER 9

# CONCLUSION

In this paper we presented Phurti, which is an application and network topology-aware scheduling framework designed for MapReduce. Phurti has interfaces both with the cluster applications to retrieve job-level traffic information and with the OpenFlow layer to learn the topology of the underlying network. We implemented and evaluated Phurti with real testbed and demonstrated the advantage of Phurti compared to application-agnostic approach. Evaluation results on real-world workloads show Phurti improves job completion time for 95% of the jobs. It decreases average job completion time by 20% for all jobs and by 23% for small jobs. It also prevents starvation by improving tail job completion time by 13%.

# REFERENCES


[1] "Apache Hadoop," http://hadoop.apache.org.

[2] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing Data Transfers in Computer Clusters with Orchestra," in *ACM SIGCOMM CCR*, vol. 41, no. 4, 2011, pp. 98–109.

[3] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data Center TCP (DCTCP)," *ACM SIGCOMM CCR*, vol. 41, no. 4, pp. 63–74, 2011.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pFabric: Minimal Near-Optimal Datacenter Transport," in *Proc. of ACM SIGCOMM*, 2013, pp. 435–446.

[5] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing Flows Quickly with Preemptive Scheduling," *ACM SIGCOMM CCR*, vol. 42, no. 4, pp. 127–138, 2012.

[6] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks," in *Proc. of USENIX NSDI*, vol. 10, 2010, pp. 19–19.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu et al., "B4: Experience with a Globally-Deployed Software Defined WAN," in *Proc. of ACM SIGCOMM*, 2013, pp. 3–14.

[8] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-Driven WAN," in *Proc. of ACM SIGCOMM*, 2013, pp. 15–26.

[9] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," in *Proc. of ACM SIGCOMM*, 2014, pp. 443–454.

[10] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized Task-Aware Scheduling for Data Center Networks," in *Proc. of ACM SIGCOMM*, 2014, pp. 431–442.

[11] H. Alkaff, I. Gupta, and L. Leslie, "Cross-Layer Scheduling in Cloud Systems," in *to appear in IEEE IC2E)*, 2015.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM CCR*, vol. 38, no. 2, pp. 69–74, 2008.

[13] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The Case for Evaluating MapReduce Performance using Workload Suites," in *Proc. of IEEE MASCOTS*, 2011, pp. 390–399.

[14] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-Gravity Reduce Task Scheduling to lower MapReduce Network Traffic," in *Proc. of IEEE CLOUD*, 2012, pp. 49–58.

[15] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou, "Optimizing Data Shuffling in Data-Parallel Computation by Understanding User-Defined Functions." in *Proc. of USENIX NSDI*, vol. 12, 2012, pp. 22–22.

[16] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Reoptimizing Data Parallel Computing," in *Proc. of USENIX NSDI*, 2012, pp. 281–294.

[17] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, and P. Lin, "Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in MapReduce Clusters," in *Proc. of IEEE SoCC*, 2013, p. 6.

[18] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in *Proc. of USENIX NSDI*, 2012, pp. 267–280.

[19] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-Resource Packing for Cluster Schedulers," in *Proc. of ACM SIGCOMM*. ACM, 2014, pp. 455–466.

[20] G. Wang, T. Ng, and A. Shaikh, "Programming your Network at Runtime for Big Data Applications," in *Proc. of ACM HotSDN*, 2012, pp. 103–108.

[21] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory Networking: An API for Application Control of SDNs," in *Proc. of ACM SIGCOMM*, 2013, pp. 327–338.

[22] A. Das, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and C. Yu, "Transparent and Flexible Network Management for Big Data Processing in the Cloud," in *Proc. of USENIX HotCloud*, 2013.

[23] A. Chanda, C. Westphal, and D. Raychaudhuri, "Content Based Traffic Engineering in Software Defined Information Centric Networks," in *Proc. of IEEE Infocom NOMEN Workshop*, 2013.

[24] M. Chowdhury and I. Stoica, "Coflow: A Networking Abstraction for Cluster Applications," in *Proc. of ACM HotNets*, 2012, pp. 31–36.

[25] "POX," https://openflow.stanford.edu/display/ONL/POX+Wiki.