AN EFFICIENT MAPREDUCE SCHEDULER FOR CLOUD ENVIRONMENT

Thesis

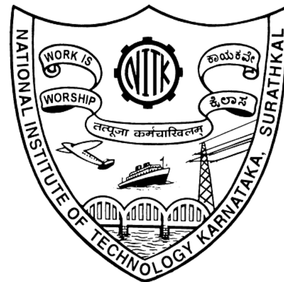
Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

by

RATHINARAJA JEYARAJ

Reg. No.: 155031 IT15F01



DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALURU - 575025

MAY 2020

DECLARATION

By the Ph.D. Research Scholar

I hereby *declare* that the Research Thesis entitled **AN EFFICIENT MAPREDUCE SCHEDULER FOR CLOUD ENVIRONMENT** which is being submitted to the **National Institute of Technology Karnataka, Surathkal** in partial fulfillment of the requirements for the award of the Degree of **Doctor of Philosophy** in **Information Technology** is a *bonafide report of the research work carried out by me*. The material contained in this Research Thesis has not been submitted to any University or Institution for the award of any degree.

Rathinaraja Jeyaraj

Reg. No.: 155031 IT15F01

Department of Information Technology

Place: NITK, Surathkal.

Date:

CERTIFICATE

This is to *certify* that the Research Thesis entitled **AN EFFICIENT MAPREDUCE SCHEDULER FOR CLOUD ENVIRONMENT** submitted by **Rathinaraja Jeyaraj**, (Reg. No.: 155031 IT15F01) as the record of the research work carried out by him, is *accepted as the Research Thesis submission* in partial fulfillment of the requirements for the award of degree of **Doctor of Philosophy**.

(Dr. Ananthanarayana V S)
Research Supervisor

Chairman - DRPC

DEDICATION AND ACKNOWLEDGMENT

It is a great opportunity to thank **Prof.Ananthanarayana V S** (Research supervisor, Department of Information Technology, National Institute of Technology Karnataka) for being constant motivation and providing valuable suggestions throughout my research journey. I express my sincere and deepest gratitude to Prof.Ananthanarayana V S, for the freedom he provided to set the research goals and pursue without any restriction. I thank the Research Progress Assessment Committee (**RPAC**) members for their continuous support and encouragement. I convey many thanks to all fellow doctoral students, teaching faculties, and non-teaching staffs in the Department of Information Technology for encouraging to pursue hardwork and their cooperation. Especially, I thank **Dr.Karthik Narasimman** (Karunya University, Coimbatore) for the timely support during many odd times and am grateful to all who helped me directly/indirectly.

It would not be an exaggeration to thank **Prof.Anand Paul** (Department of Computer Science and Engineering, Kyungpook National University, Korea) for providing me an opportunity to work in his lab for six months and for his valuable suggestions to shape my research work.

It is always impossible without the family support to invest huge time for research. I am debted to my parents, **Mrs.Radha Ambigai Jeyaraj** and **Mr.Jeyaraj Rathinasamy**, for the whole life. I also thank my brothers **Mr.Sivaraja Jeyaraj** and **Mr.Elayaraja Jeyaraj** for supporting me financially without any expectation. Life would be incomplete without friends **Mr.Benjamin Santhosh Raj** and **Mr.Rajkumar Rathinam** for spending their lovely time and fun talk. Finally, I should mention my source of inspiration right from graduate studies, my wife, **Dr.Sujiya Rathinaraja**, who consistently gave mental support all through the tough journey. Infinite thanks to her for keeping my life green and lovable.

ABSTRACT

Hadoop MapReduce is one of the cost-effective ways to process a large volume of data for reliable and effective decision-making. As on-premise Hadoop cluster is not affordable for short-term users, many public cloud service providers like Amazon, Google, and Microsoft typically offer Hadoop MapReduce and relevant applications as a service via a cluster of virtual machines over the Internet. In general, these Hadoop virtual machines are launched in different physical machines across cloud data-center and co-located with non-Hadoop virtual machines. It introduces many challenges, more specifically, a layer of heterogeneities (hardware heterogeneity, virtual machine heterogeneity, performance heterogeneity, and workload heterogeneity) that impacts the performance of MapReduce job and task scheduler. Containing physical servers of different configuration and performance in cloud data-centers is called hardware heterogeneity. Existence of different size of virtual machines in a Hadoop virtual cluster is called virtual machine heterogeneity. Hardware heterogeneity, virtual machine heterogeneity, and co-located non-Hadoop virtual machine's interference together cause varying performance for the same map/reduce task of a job. This is called performance heterogeneity. Latest MapReduce versions allow users to customize the resource capacity (container size) for the map/reduce tasks of different jobs. This leads a batch of MapReduce of jobs to be heterogeneous.

These heterogeneities are inevitable and profoundly affect the performance of MapReduce job and task scheduler concerning job latency, makespan, and virtual resource utilization. Therefore, it is essential to exploit these heterogeneities while offering Hadoop MapReduce as a service to improve MapReduce scheduler performance in real-time. Existing MapReduce job and task schedulers addressed some of these heterogeneities but fell short in improving the performance. In order to improve these qualities of service further, we proposed a following set of methods: Dynamic Ranking-based MapReduce Job Scheduler (DRMJS) to exploit performance heterogeneity, Multi-Level Per Node Combiner (MLPNC) to minimize the number of intermediate records in the shuffle phase, Roulette Wheel Scheme (RWS) based data block placement and a constrained 2-dimensional bin packing model to exploit virtual machine and workload level hetero-

geneities, and Fine-Grained Data Locality Aware (FGDLA) job scheduling by extending MLPNC for a batch of jobs.

Firstly, DRMJS is proposed to improve MapReduce job latency and resource utilization by exploiting heterogeneous performance. The DRMJS calculates the performance score for each Hadoop virtual machine based on CPU and Disk IO for map tasks, CPU and Network IO for reduce tasks separately. Then, a rank list is prepared for scheduling map tasks based on map performance score, and reduce tasks based on reduce performance score. Ultimately, DRMJS improved overall job latency, makespan, and resource utilization up to 30%, 28%, and 60%, respectively, on average compared to existing MapReduce schedulers. To improve job latency further, MLPNC is introduced to minimize the number of intermediate records in the shuffle phase, which is responsible for the significant portion of MapReduce job latency. In general, each map task runs a dedicated combiner function to minimize the number of intermediate records. In MLPNC, we split the combiner function from map task and run a single MLPNC in every Hadoop virtual machine for a set of map tasks of the same job. These map tasks write its output to the common MLPNC, which minimizes the number of intermediate records level by level. Ultimately, MLPNC improved job latency up to 33% compared to existing MapReduce schedulers for a single job. However, in production environment, a batch of MapReduce jobs is periodically executed. Therefore, to extend MLPNC for a batch of jobs, we introduced FGDLA job scheduler. Results showed that FGDLA minimized the amount of intermediate data and makespan up to 62.1% and 32.4% when compared to existing schedulers.

Secondly, virtual machine and workload level heterogeneities cause resource underutilization in the Hadoop virtual cluster and impact makespan for a batch of MapReduce jobs. Considering this, we proposed RWS based data block placement, and a constrained 2-dimensional bin packing to place heterogeneous map/reduce tasks onto heterogeneous virtual machines. RWS places data blocks based on the processing capacity of each virtual machine, and bin packing model helps to find the right combination of map/reduce tasks of different jobs for each bin to improve makespan and resource utilization. The experimental results showed that the proposed model improved makespan

and resource utilization up to 57.9% and 59.3% over MapReduce fair scheduler.

KEYWORDS: Bin Packing; Combiner; Heterogeneous Performance; Heterogeneous MapReduce Workloads; MapReduce Job Scheduler; MapReduce Task Placement.

Contents

Abstract i
List of Figures viii
List of Tables xii
Abstract i

1 INTRODUCTION 1

1.1 Big data and Hadoop 1
1.2 MapReduce Job 2
1.3 MapReduce Job Execution Sequence 4
1.4 MapReduce on Cloud 6
1.4.1 Heterogeneity for MapReduce on cloud 7
1.4.2 Resource usage of the MapReduce execution sequence 8
1.4.3 Dynamic/Heterogeneous performance of VMs 10
1.4.4 Heterogeneous VMs and heterogeneous MapReduce workloads 12
1.5 Research Motivation 15
1.6 Outline of the Thesis 15

2 Literature Survey and Proposed Works 17

2.1 Literature Survey 17
2.1.1 MapReduce job and task scheduling in a virtualized heterogeneous environment 17
2.1.2 Scheduling reduce tasks based on its input size 21
2.1.3 Minimizing the size of intermediate data during the shuffle phase in a virtual environment 22
2.1.4 Block placement schemes in HDFS 26
2.1.5 Bin packing tasks 27
2.2 Key Observations 27

2.2.1 MapReduce job and task scheduling in a virtualized heterogeneous environment . . . 27
2.2.2 Scheduling reduce tasks based on its input size . . . 28
2.2.3 Minimizing the size of intermediate data during the shuffle phase . . . 28
2.2.4 Block placement schemes in HDFS . . . 29
2.2.5 Bin packing tasks . . . 29
2.3 Problem Definition . . . 29
2.4 Research Objectives and Works . . . 29
3 MapReduce Task Scheduling **31**
3.1 Proposed Methodologies . . . 31
3.1.1 Dynamic Ranking based MapReduce Job Scheduler (DRMJS) . . . 32
3.1.2 Map and reduce task scheduling based on performance rank . . . 34
3.1.3 Scheduling reduce tasks based on its input size . . . 37
3.1.4 Multi-Level Per Node Combiner (MLPNC) . . . 42
3.2 Results and Analysis . . . 46
3.2.1 Dynamic Ranking based MapReduce Job Scheduler (DRMJS) . . . 46
3.2.2 Multi-Level Per Node Combiner (MLPNC) . . . 55
3.2.3 Reduce task scheduling based on performance rank after MLPNC 57
3.3 Summary . . . 62
4 MapReduce Job Scheduling **65**
4.1 Proposed Methodologies . . . 65
4.1.1 Roulette Wheel Scheme (RWS) based data block placement . . . 65
4.1.2 Constrained 2-dimensional bin packing map/reduce tasks . . . 68
4.1.3 Packing map/reduce tasks using Ant Colony Optimization (ACO) 73
4.1.4 Fine Grained Data Locality Aware (FGDLA) job scheduling . . . 76
4.2 Results and Analysis . . . 79
4.2.1 Bin packing map/reduce tasks using ACO . . . 79
4.2.2 Fine-Grained Data Locality-Aware scheduler (FGDLA) . . . 84
4.3 Summary . . . 89

5 Conclusion and Future Work **91**
5.1 Conclusion 91
5.2 Future Work 92
6 Appendix **93**
6.1 Course Work 93
6.2 Work Timeline 93
6.3 List of Publications 95
6.3.1 International Journals 95
6.3.2 International Conferences 95
6.4 References 97

List of Figures

1.1 Hadoop MapReduce v2 cluster 3
1.2 MapReduce phases 3
1.3 Containers 4
1.4 MapReduce execution sequence 5
1.5 Hadoop VMs deployed in CDC 7
1.6 Heterogeneity in different layers 8
1.7 Disk and network IO consumption by map and reduce task for wordcount job 9
1.8 CPU usage by map and reduce task for wordcount job 10
1.9 Disk IO consumption of map task for wordcount job during co-located VM's interference 11
1.10 IO (Disk and N/W) consumption for reduce task in wordcount job during co-located VM's interference 11
1.11 Map and reduce task latency variation on different class of PMs for wordcount job 13
1.12 Unused CPU and N/W resources due to Disk IO contention in each PM for map task 13
1.13 Heterogeneous workloads, VMs, PMs 14
1.14 Task scheduling with/without heterogeneous capacity 14

2.1 Default combiner 23
2.2 Per Node Combiner (PNC) [11] 24

3.1 VMs sharing resources in a PM 33
3.2 Workflow of DRMJS 42
3.3 MLPNC 43

3.4 Storing intermediate records in Memcache 45
3.5 MLPNC system architecture 45
3.6 Average map/reduce task latency of wordcount job with different cases . 49
3.7 Average map/reduce task latency of sort job with different cases 49
3.8 Average map/reduce task latency of wordmean job with different cases . 50
3.9 Job latency of different workloads with different cases 50
3.10 Makespan of different cases 50
3.11 Average reduce task latency of Case 3 and Case 4 for wordcount job . . 52
3.12 Performance score vs number of map/reduce tasks allocated 53
3.13 Resource utilization after DRMJS 54
3.14 Number of shuffled records generated by different approaches for different sizes of dataset 56
3.15 Average shuffle latency using different approaches for different sizes of dataset 56
3.16 Reduce task start latency for all datasets based on different approaches 57
3.17 Overall job latency 57
3.18 Case 1: Reduce task latency with no combiner 59
3.19 Case 2: Reduce task latency with combiner 61
3.20 Reduce task latency with dynamic performance vs MLPNC 62

4.1 Number of map/reduce task combinations of different jobs 73
4.2 Finding map/reduce task combinations using ACO 74
4.3 Example using FGDLA 77
4.4 FCFS vs FAIR vs FGDLA 78
4.5 Latency of jobs using different schedulers 81
4.6 Number of non-local executions 81
4.7 Makespan 83
4.8 Utilization of the vCPU 83
4.9 Utilization of the memory 83
4.10 Average resource wastage of different schedulers 84
4.11 Resource requirements of each job 85

4.12 Latency of jobs 86
4.13 Number of non-local executions 86
4.14 Size of shuffle data 87
4.15 Makespan, number non-local executions, shuffle data size of each job . 88
4.16 Unused number of vCPUs during execution 88
4.17 Unused memory during execution 89

List of Tables

1.1 Physical machines configuration 12

3.1 Performance class 38
3.2 Physical Machines (PM) configuration 46
3.3 Hadoop virtual cluster 47
3.4 Average map/reduce task latency for different workloads on heterogeneous environment 48
3.5 Number of reduce tasks and its average latency for wordcount job using Case 3 and Case 4 51
3.6 Avoidance of map/reduce tasks from interference 54
3.7 Number of reduce tasks for all cases 59
3.8 Number of reduce tasks and its average latency 60

4.1 Percentage of blocks to store in different VM Flavours 66
4.2 Possible combination of map tasks of different jobs in a VM 71
4.3 Maximum number of map tasks for each job in each VM flavour 74

ABBREVIATIONS

ACO	Ant Colony Optimization
CDC	Cloud Data-Center
CSP	Cloud Service Provider
DRMJS	Dynamic Ranking based MapReduce Job Scheduler
FGDLA	Fine Grained Data Locality Aware
HDD	Hard Disk Drive
HDFS	Hadoop Distributed File System
IaaS	Infrastructure as a Service
IO	Input/Output
IS	Input Split
LFS	Local File System
MLPNC	Multi Level Per Node Combiner
NIC	Network Interface Card
MRAppMaster	MapReduce Application Master
OCRU	Overall Cluster Resource Utilization
PM	Physical Machines
PNC	Per Node Combiner
QoS	Quality of Services
RR	Record Reader
RW	Record Writer
RWS	Roulette Wheel Scheme
TAR	Total Allocated Resource
UIB	Utilization of Individual Bin
VM	Virtual Machine
YARN	Yet Another Resource Negotiator

Chapter 1

INTRODUCTION

Big data analytics [1] using Hadoop MapReduce on cloud [2] by small scale enterprises, research departments, and educational institutions is increasingly becoming popular. As on-premise Hadoop cluster is not affordable for short-term users, public Cloud Service Providers (CSP) like IBM, Microsoft, Amazon, and Google offer MapReduce and relevant applications as a service. Therefore, end users make use of them without any up-front capital investment for on-premise IT infrastructure and software licensing by leveraging cloud's cost-efficient, scalable on-demand service nature. CSP typically offer MapReduce via a cluster Virtual Machines (VM), which are placed across Cloud Data Center (CDC) and co-located with non-Hadoop VMs. This introduces many challenges for Hadoop MapReduce to face in a virtualized environment. More specifically, heterogeneity impacts the performance of MapReduce job and task schedulers in a virtualized environment. This thesis investigates various heterogeneities that exist while offering Hadoop MapReduce as a service and proposes a set of methods to improve job latency, makespan, and resource utilization.

1.1 Big data and Hadoop

Any characteristic (volume/velocity/variety/value/variability/complexity) of data that outpaces the storage capacity, computation capability, and algorithm ability in a machine is called big data [3]. Processing big data helps to increase productivity in business, improve operational efficiency in management, and get insight (knowledge) in scientific research. There are various big data processing frameworks: Hadoop, Stratosphere, Spark, Storm, etc. Hadoop is one of the best batch processing tools to store and

process huge amount of data using a cluster of unreliable, low-cost commodity servers. Hadoop includes three primary tools to batch process big data: Hadoop Distributed File System (HDFS), Yet Another Resource Negotiator (YARN), and MapReduce. HDFS is used to store and retrieve big data from distributed storage on a cluster of servers. It breaks massive data into equal sized chunks (called blocks) and stores in different slave nodes with the desired number of replication. Hadoop MapReduce [4] is a distributed data parallel programming model to crunch huge data uploaded onto HDFS. MapReduce is highly distributed, horizontally scalable, fault tolerant, high throughput, and flexible software programming model that helps to write scalable algorithms to process big data stored, typically, in HDFS. YARN is a centralized cluster resource manager to share cluster resources and data stored in HDFS among different data frameworks. The sub-components of these three tools are:

- YARN: Resource Manager (RM), Node Manager (NM)
- HDFS: Name Node (NN), Secondary Name Node (SNN), Data Node (DN)
- MapReduce v2: MapReduce Application Master (MRAppMaster), Job History Server (JHS)

As shown in Figure 1.1, consider a CDC containing four racks each with four Physical Machines (PM), denoted as nodes, and two MapReduce jobs. A typical Hadoop cluster on a physical server cluster is shown in Figure 1.1. Each master sub-component (NN, RM, JHS) is installed in a dedicated node and slave sub-components (DN, NM) are installed in all other nodes in the cluster. There are two MRAppMaster components, as two MapReduce jobs are running.

1.2 MapReduce Job

A MapReduce job consists of two phases: map phase, and reduce phase. Map phase executes a set of map tasks and reduce phase executes a set of reduce tasks of a job. Shuffle in reduce phase transfers output of all map tasks from map phase to all the reduce tasks in reduce, as shown in Figure 1.2. Map and reduce tasks may be executed

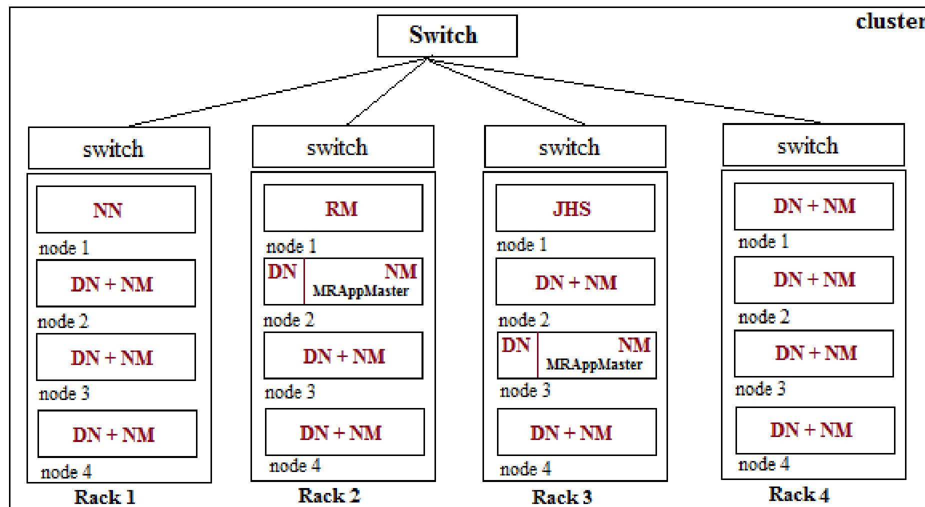


Figure 1.1 Hadoop MapReduce v2 cluster

in any nodes in the cluster. For instance, consider a MapReduce job with four map tasks and two reduce tasks. map task 1 and map task 2 are executed in node 1, but map task 3 is executed in node 2 and map task 4 is executed in node 3. Similarly, both the reduce tasks are executed in different nodes. Map phase starts when the first map task is executed and ends when all the map tasks of a job is completed. Similarly, reduce phase starts when the intermediate data is moved to reduce node, however, a reduce task is executed only after shuffle is completed.

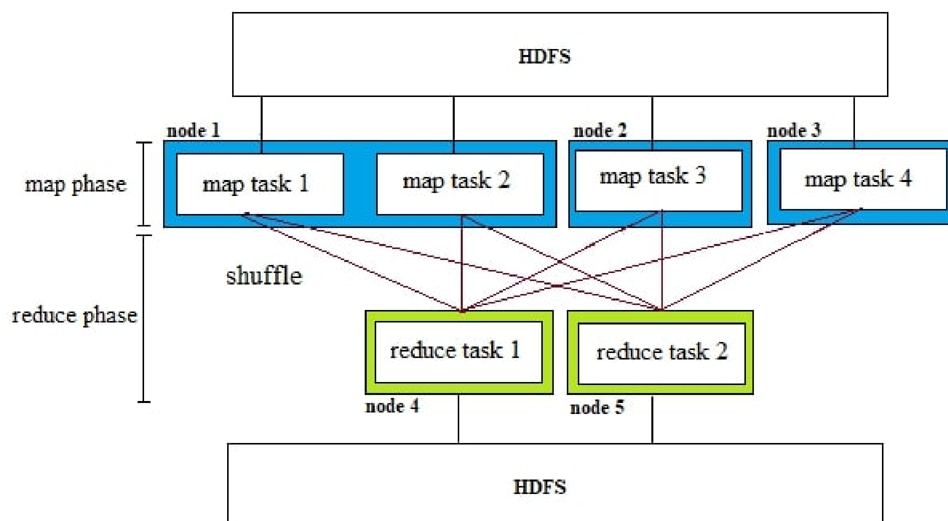


Figure 1.2 MapReduce phases

Each map task is given one or more data blocks from HDFS as input. A map task reads data from data blocks as records (key:value pairs), typically performs a record-level transformation such as filtering/extracting relevant fields, and produces an arbitrary number of records as intermediate output (map output). These intermediate records are moved to reduce nodes (server running reduce tasks) over the network. Each reduce task receives input (intermediate records) from all map tasks and produces the final output records onto HDFS. Map and reduce tasks are assigned with a resource unit, called container, for execution. A container is a logical pack of a small portion of memory and vCPU (typically virtual cores C1, C2, etc.), as shown in Figure 1.3. A node is capable of holding more than one container depending on the amount of resource available.

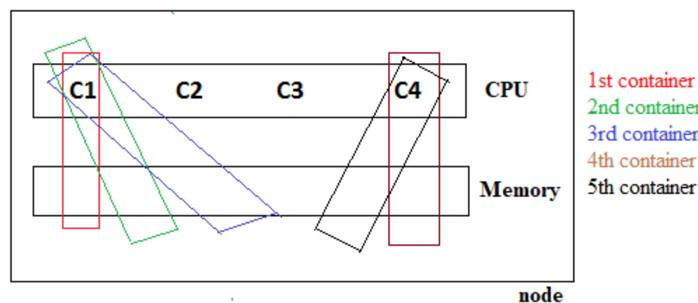


Figure 1.3 Containers

1.3 MapReduce Job Execution Sequence

Map task and reduce task go through a sequence of steps (Figure 1.4) to carry out data processing. Input and output of every step are based on key-value pairs (records). Major steps are given below.

Loading input file onto HDFS: Input file → blocks

Initially, the input file is loaded onto HDFS by dividing into equal sized chunks, called blocks. Three copies of each block are prepared by default and stored onto HDFS to ensure fault tolerance.

Map phase: file input format → Input Split (IS) → Record Reader (RR) → mapper → partitioner → combiner

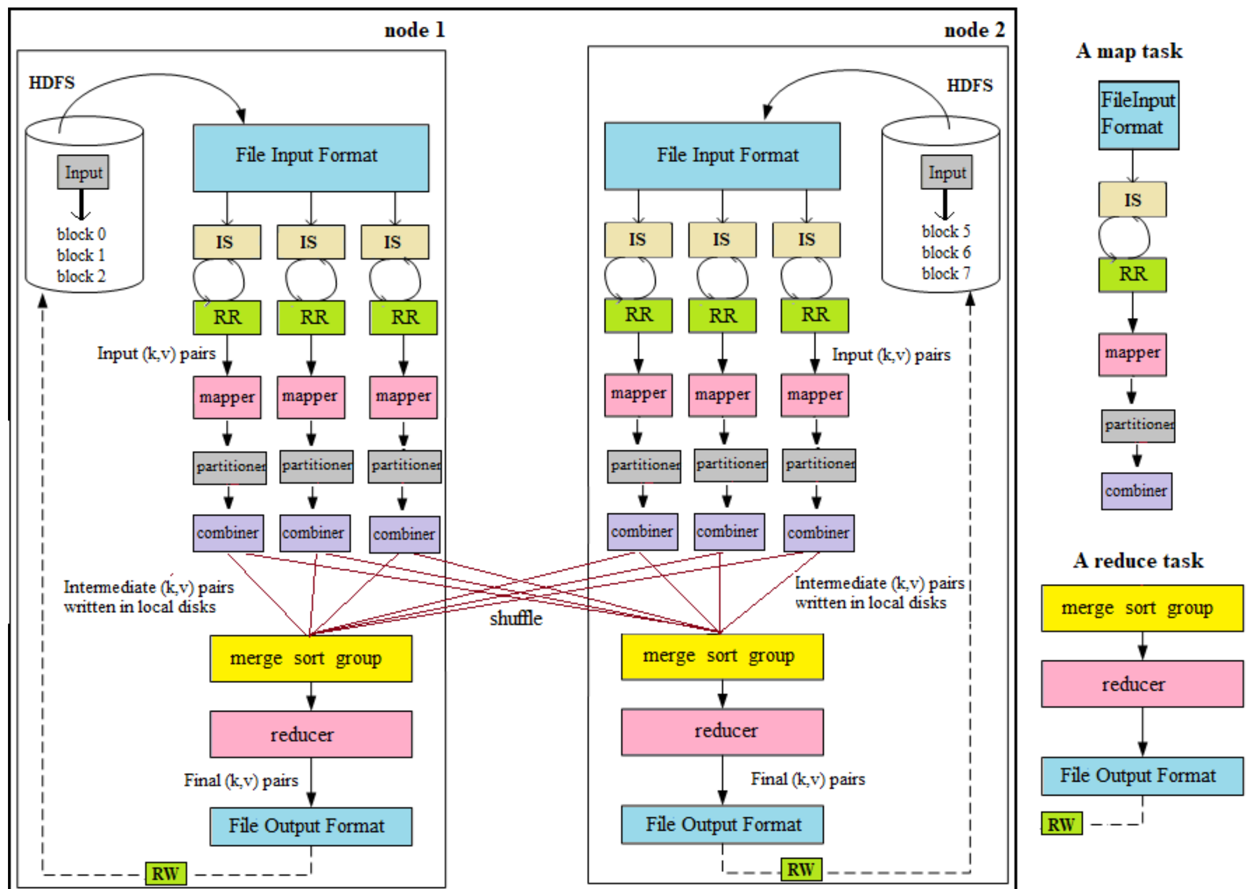


Figure 1.4 MapReduce execution sequence

Once a MapReduce job is launched on the input file, one or more blocks is given as input (called as IS) to each map task. The file input format prepares the records (key:value pairs) from IS. The RR converts byte-oriented input records to MapReduce data type and feeds to the mapper function. Mapper is an user-defined function that takes a record as input and produces an arbitrary number records as output, which are called as intermediate output and locally stored in in-memory buffer (by default 100 MB). Once buffer capacity reached a threshold, a partitioner function splits the intermediate output into a number of partitions, which is equal to the number of reduce tasks assigned for the job, and stores them into the local disk. There can be many spills over time in the local disk. Once a map task completed, the spilled partitions for respective reduce tasks are merged, and moved to the reduce nodes. If the output of map task is huge, transferring them to reduce node over the network will introduce more traffic.

So, the combiner function is applied to minimize the number of intermediate records. Both combiner and partitioner functions are optional and can be defined by the users. Executing map task in a node where the data block available is called data locality.
Reduce phase: shuffle → merge → sort → group → reducer → File Output Format → Record Writer (RW).
Shuffle moves the output of all map tasks over the network to the nodes running respective reduce tasks.A reduce task receives its inputs from all map tasks and merges them. Then, merged records are sorted based on key, and the values that belong to the same key are grouped. MapReduce framework takes care of these magical steps (shuffle, sort, merge, group). Finally, reducer function processes these grouped records (key:list(values)) and produces an arbitrary number of records as job output onto HDFS by RW based on file output format.
Output file onto HDFS: RW writes the arbitrary number of output records onto HDFS based on file output format. If the output file size goes beyond default block size, then it is divieded into multiple blocks.

1.4 MapReduce on Cloud

On-premise IT infrastructure for Hadoop MapReduce is not affordable for short-term users. Therefore, CSP offers MapReduce and relevant applications as a service on-demand for pay-per-use basis. CSP deliver MapReduce service to end users via Infrastructure as a service (IaaS) with different flavors, as given below.

- Private Hadoop MapReduce (pay per VM or PM hired).
  - Purchase VMs or PMs from CSP and setup MapReduce manually.
  - Purchase MapReduce as a service on a cluster of PMs or VMs.
- Sharing MapReduce service with more than one users (pay per job basis).

Obtaining MapReduce as a service on a cluster of VMs is highly scalable and based on pay-per-use basis. Therefore, short-term users highly prefer this service from cloud. As shown in Figure 1.5, VMs (shaded) in Hadoop virtual cluster are typically spread across the CDC to achieve fault tolerance and co-located with non-Hadoop VMs. It introduces

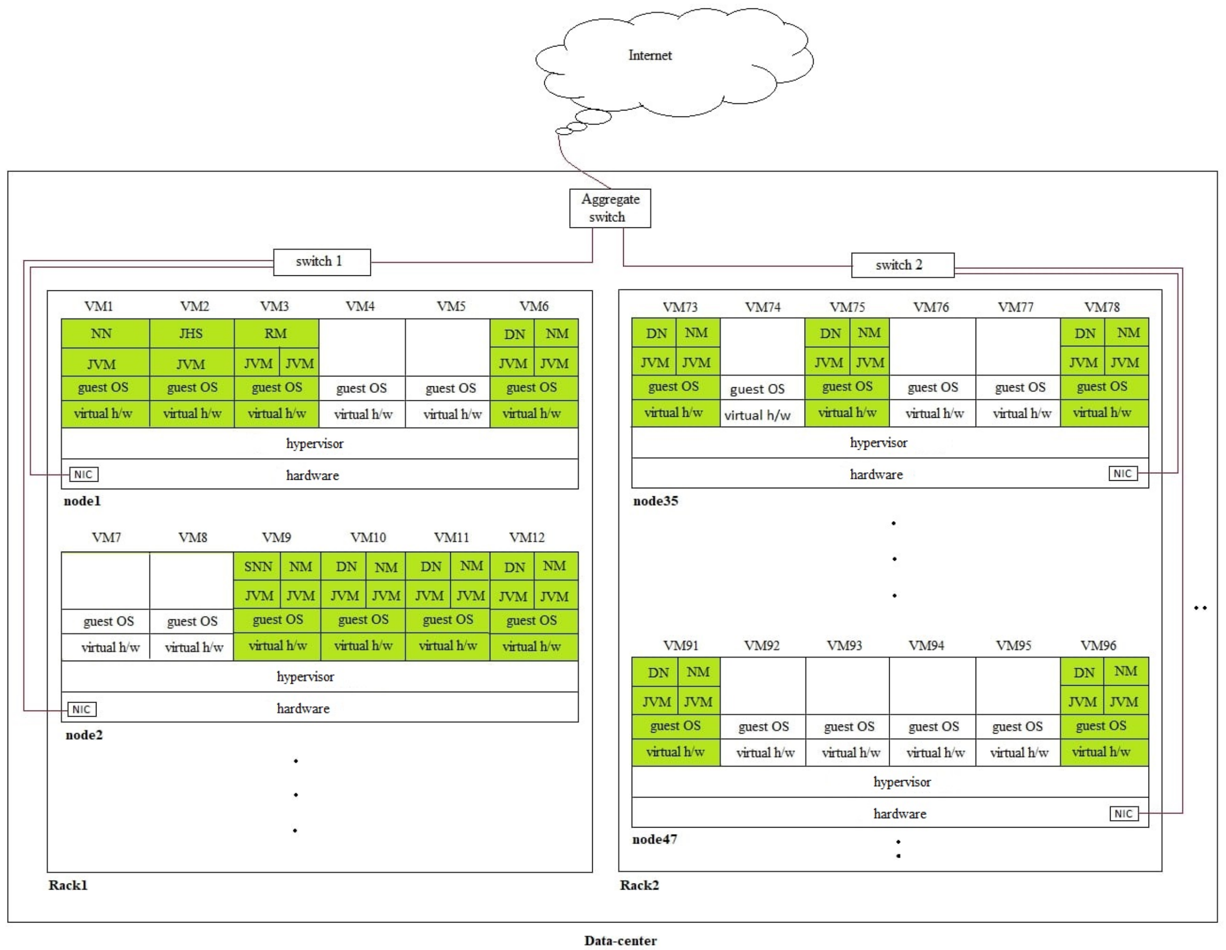


Figure 1.5 Hadoop VMs deployed in CDC

many challenges, more specifically, a layer of heterogeneities (hardware heterogeneity, virtual machine heterogeneity, performance heterogeneity, and workload heterogeneity) impacts the performance of MapReduce job and task scheduler. In the subsequent sections, these heterogeneities and its impact on the performance of MapReduce job and task schedulers are discussed.

1.4.1 Heterogeneity for MapReduce on cloud

In general, heterogeneity is the characteristics of containing dissimilar elements. Heterogeneity may exist [5] in CPU, storage, and network resources in a CDC. Typically, CDC is composed with different types of computing, storage, and network components to support a wide range of workloads and various user needs. A layer of heterogeneity

[6] is identified while offering MapReduce on a cluster of VMs from cloud, as shown in Figure 1.6.

- Hardware heterogeneity: All inter-connected physical servers are not of the same configuration and performance. Hybrid cloud involves large hardware heterogeneity.
- VM heterogeneity: A cluster of VMs allocated for MapReduce may not be the same size (flavor) due to horizontal scaling service nature.
- Performance heterogeneity: VM performance varies dynamically due to hardware heterogeneity, VM heterogeneity, and co-located VM's interference.
- Workload heterogeneity: This indicates jobs of different size (number of map and reduce tasks), job latency, resource requirement (container size), the size of data to process, etc.,

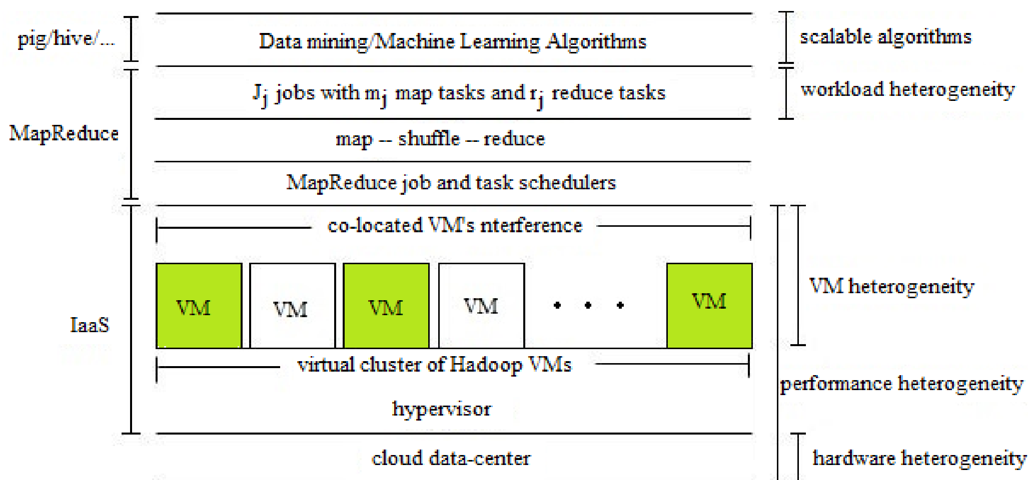


Figure 1.6 Heterogeneity in different layers

1.4.2 Resource usage of the MapReduce execution sequence

A MapReduce job consumes varying resource during execution sequence. In this section, we give some experimental evidence on map and reduce tasks resource consumption behavior. To demonstrate resource usage during execution, we used wordcount

job on PUMA Wikipedia dataset [7]. *collectl* tool is used to monitor CPU, Disk, network (N/W) consumption and MapReduce counters to get HDFS IO related metrics. For wordcount job, map task takes more Disk IO and CPU, while reduce task takes more of CPU and N/W IO (Figure 1.7). During shuffle phase, N/W bandwidth is highly consumed in a particular area of the virtual cluster. To differentiate HDFS access with local file system access, we split Disk IO into Local File System (LFS) IO and HDFS IO. Because HDFS is used only when input blocks are read, and output blocks are written, whereas LFS is used to spill intermediate results of the map and shuffled records in reduce tasks.

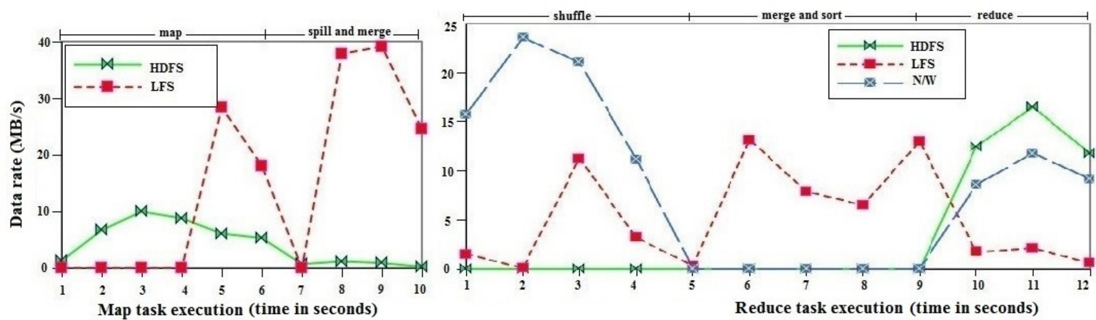


Figure 1.7 Disk and network IO consumption by map and reduce task for wordcount job

From Figure 1.7, we can observe that map tasks use more Disk IO than reduce tasks. Because, initially, HDFS brings data blocks for map tasks into memory on demand; later, intermediate records are spilled into LFS. At the time of merging intermediate files, LFS is heavily used for preparing partitions to send to reduce tasks. In the reduce phase, shuffle involves a lot of N/W IO as every reduce task fetches its input from all map tasks. LFS is accessed in reduce phase for merge and sort operations. Network IO is used over 65% than Disk IO for reduce tasks. From Figure 1.8, we can observe that CPU is used more for reduce tasks compared to map tasks. Because, in the map phase, only pre-processing records/merge operations are carried out. However, major algorithm implementations, sorting, grouping operations are carried out in the reduce phase, which demands 60% of CPU compared to map tasks. Default schedulers (FIFO, Capacity, Fair) place map and reduce tasks upon resource availability (resource-aware).

Being resource-aware vastly reduces resource utilization in the virtual cluster. For example, if disk IO bottleneck is encountered in a node, map task keeps CPU idle for a long time as resource allocation for tasks is space sharing. Therefore, instead of launching a map task at this moment, reduce task can be launched to make use of CPU and N/W IO.

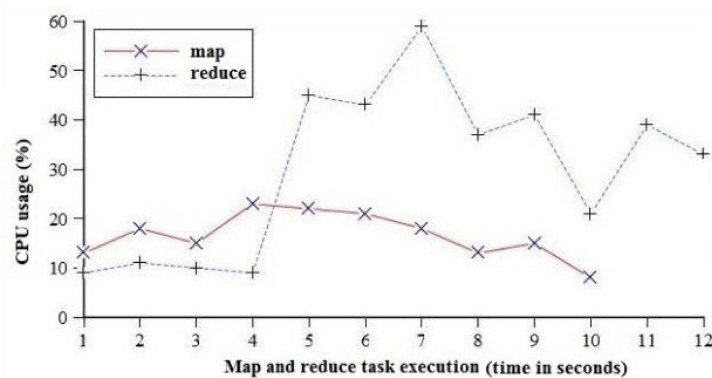


Figure 1.8 CPU usage by map and reduce task for wordcount job

1.4.3 Dynamic/Heterogeneous performance of VMs

Every PM in CDC hosts a set of VMs. A Hadoop virtual cluster is launched across racks in different PMs to ensure fault tolerance. This causes to different latency for the same task. Hadoop VMs are typically co-located with non-Hadoop VMs, where CPU and memory are space-shared while IO (N/W and disk) is time-shared for VMs. Therefore, co-located VMs cause varying performance for the same task due to VM's competition to hold shared resources on demand. To experiment this, we launched a Hadoop VM along with non-Hadoop VM in a PM and run wordcount job to display how resources are consumed during interference. We introduced random read and write to disk and N/W (for IO contention) via non-Hadoop VMs. It largely impacted map and reduce tasks latency, as evidenced in Figure 1.9 and Figure 1.10. Figure 1.9 shows Disk IO consumption of map task with four cases: HDFS access with interference, HDFS access with no interference, LFS access with interference, and LFS access with no interference. It is evident that map latency of wordcount job increased up to 50% than map task execution with no interference. Similarly, Figure 1.10 shows IO (disk

and N/W) consumption of reduce task with six cases: HDFS access with interference, HDFS access with no interference, LFS access with interference, LFS access with no interference, N/W access with interference, and N/W access with no interference. We can observe that reduce task latency increased over 60% due to co-located VM's race on shared resources.

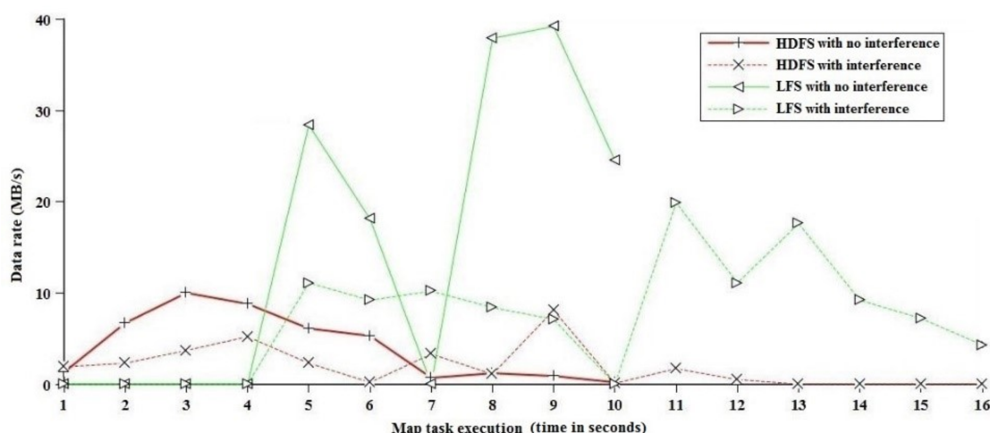


Figure 1.9 Disk IO consumption of map task for wordcount job during co-located VM's interference

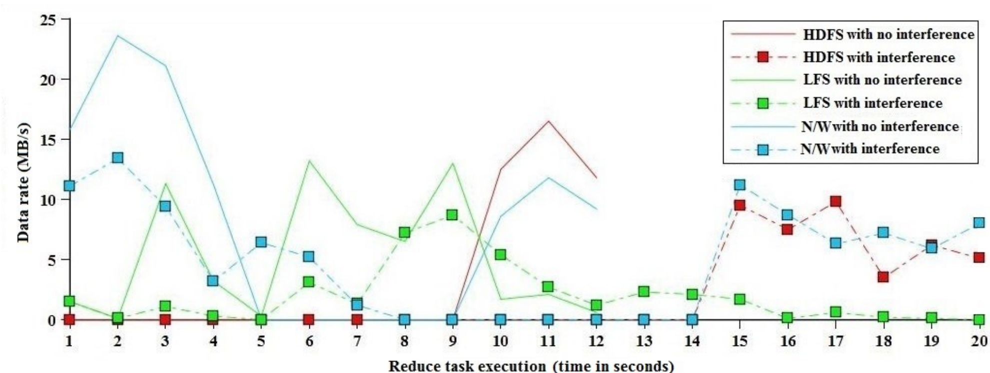


Figure 1.10 IO (Disk and N/W) consumption for reduce task in wordcount job during co-located VM's interference

From Figure 1.9, it is evident that the map task uses disk IO for long time due to co-located VM's interference and holds CPU idle. It directly increased job latency and also leads to resource under-utilization. Similarly, Figure 1.10 illustrates that reduce

task uses N/W IO for quite some time due to co-located VM's demand on a N/W resource. Therefore, not only map/reduce task latency increased and also resources were held idle until task completion. Finally, PMs that host VMs are of different configurations and capacities in a cluster, as given in Table 1.1. To experiment heterogeneous performance and the impact of IO contention placed by non-Hadoop VMs in a different class of PMs, we run wordcount job in a VM of each PM class. As shown in Figure 1.11, both map and reduce task's latency vary considerably depending on the performance of PMs. The interesting fact is that, when IO contention is on a roll, CPU and memory are held idle by map task. For instance, PM_4 in Figure 1.12 shows that when Disk IO contention is encountered due to co-located non-Hadoop VMs, CPU and N/W resources of Hadoop VM are mostly unused. This indicates that hired virtual resources are not utilized beneficially and job latency increased consequently. Varying resource consumption behaviour of map/reduce tasks and heterogeneous performance suggest that just allocating map/reduce tasks in a VM increased job latency and caused resource under-utilization in a heterogeneous, virtualized environment. This motivated us to exploit underlying hardware heterogeneity and co-located VM's interference.

1.4.4 Heterogeneous VMs and heterogeneous MapReduce workloads

For both map and reduce tasks, a container is allocated for execution, which is represented as $< vCPU, Memory >$. The container is flexible and can be defined for each job. For instance, from Figure 1.13, there are six jobs (J_i | i=1 to 6), whose map and reduce tasks demand different size of containers. For example, J_1 demands 1 vCPU and 2 GB

Table 1.1 Physical machines configuration

PMs class	Configuration of PMs
PM_1	1.90GHz 6 cores, 32 GB memory, 1 TB HDD
PM_2	2.40GHz 28 cores, 132 GB memory, 3 TB HDD
PM_{3-4}	3.20GHz 4 cores, 8 GB memory, 1 TB HDD
PM_{5-8}	3.40GHz 4 cores, 8 GB memory, 1 TB HDD

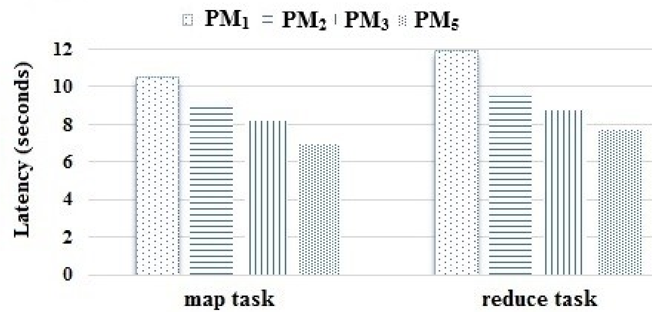


Figure 1.11 Map and reduce task latency variation on different class of PMs for word-count job

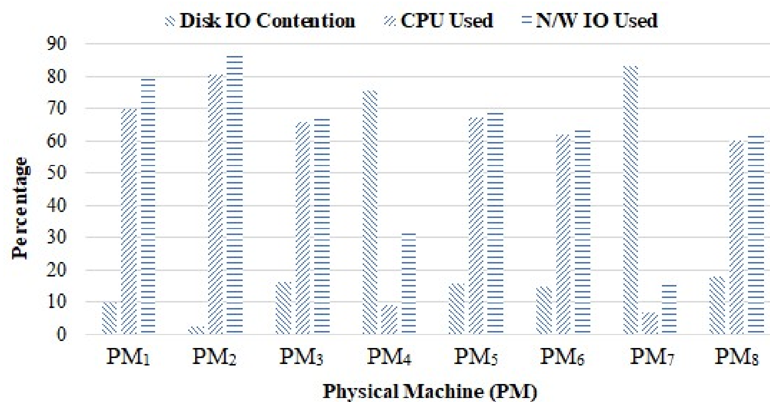


Figure 1.12 Unused CPU and N/W resources due to Disk IO contention in each PM for map task

memory for map tasks, and 1 vCPU and 1 GB memory for reduce tasks. A number of map and reduce tasks, and job nature also vary in each job. However, such flexibility introduced challenges to be addressed to improve makespan and resource utilization as heterogeneous jobs are periodically submitted as a batch. Even though CSP offer unlimited virtual resources, it is questionable whether all the hired virtual resources are utilized maximum at any point of time during service lifetime. In a rough estimation, if a VM wastes 0.5 GB memory in the cluster of 200 VMs, net wastage is 100 GB. This primarily affects cloud users to pay for unused capacity over a period. Such resource under-utilization happens due to many reasons for different applications. VMs deployed for MapReduce may be of different size (flavor) [5],[6] and causes varying number of containers to accomodate.

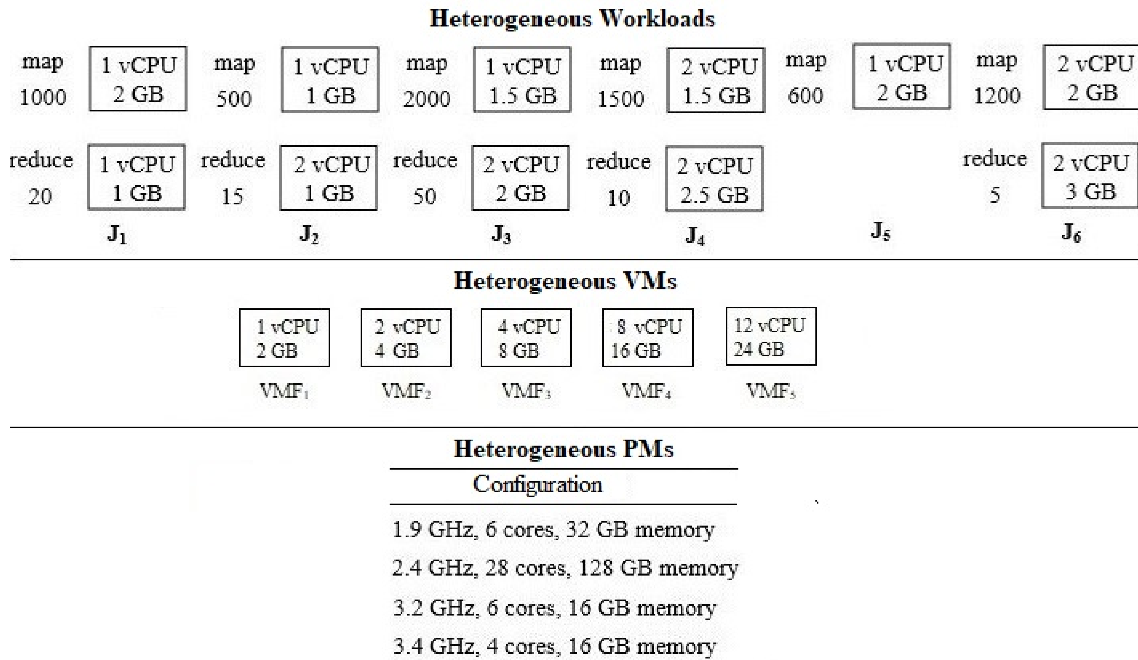


Figure 1.13 Heterogeneous workloads, VMs, PMs

For instance, as shown in Figure 1.14, consider two VMs with different configuration of $< vCPU, Memory >:< 4,6 >$, and $< 2,4 >$ respectively. As shown in Figure 1.14(a), if map tasks of J_4 are scheduled in VM_1, only two map tasks can be placed resulting to 3 GB unused memory until running map tasks finished execution. Similarly, if a map task from J_3 and J_2 is scheduled in VM_2, 1.5 GB memory is not utilized until these map tasks get over. However, if we schedule map tasks of different jobs understanding the capacity of VMs, it is possible to minimize the hired resource wastage. As shown in Figure 1.14(b), if we launch a map task of J_1, J_5, J_6 in VM_1 and J_1, J_5 in VM_2, it is possible to utilize the entire hired virtual resources. It is also true for reduce tasks of

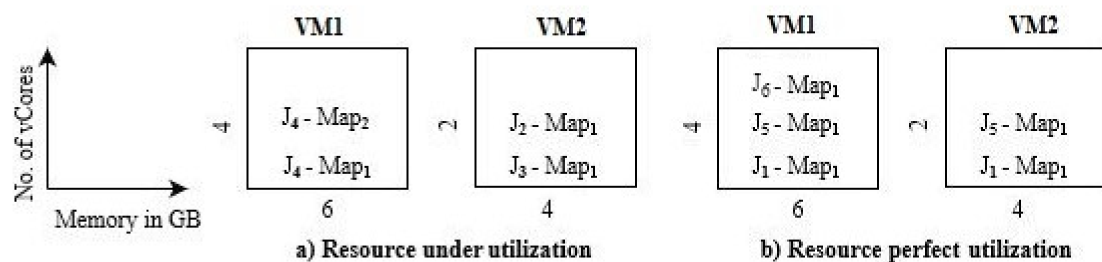


Figure 1.14 Task scheduling with/without heterogeneous capacity

different jobs demanding the varying size of the containers. Therefore, scheduling map and reduce tasks of different jobs in the right combination improves resource utilization of individual VMs. Therefore, it is essential to exploit these heterogeneities while offering Hadoop MapReduce as a service to improve MapReduce scheduler performance in real-time.

1.5 Research Motivation

Offering MapReduce as a service faces many challenges in a virtualized environment. More significantly, heterogeneity that exists at various level while offering MapReduce as a service impacts MapReduce task and job schedulers performance. This motivated us to develop an efficient MapReduce task and job scheduler to improve job latency, makespan, and resource utilization in a heterogeneous virtualized environment.

1.6 Outline of the Thesis

Related works on MapReduce task and job scheduling in a heterogeneous virtualized environment for heterogeneous workloads are discussed in Chapter 2. Subsequently, research objectives are set considering the shortcomings identified. Chapter 3 presents a set of proposed methods for MapReduce task scheduler to improve job latency and resource utilization, while Chapter 4 accounts the proposed methods for MapReduce job scheduler to improve makespan and resource utilization. Conclusion and future works are mentioned in Chapter 5.

Chapter 2

Literature Survey and Proposed Works

2.1 Literature Survey

MapReduce job/task scheduling is very challenging in a virtualized environment while offering as a service. Notably, different levels of heterogeneity is unavoidable and must be addressed to improve MapReduce job latency, makespan, and virtual resource utilization. Some of the prominent works are discussed below in order to emphasize this thesis importance. This section surveys the existing works on the following topics:

1. MapReduce job and task scheduling in a virtualized heterogeneous environment.

2. Scheduling reduce tasks based on its input size.

3. Minimizing the size of intermediate data during the shuffle phase.

4. Block placement schemes in HDFS.

5. Bin packing tasks.

2.1.1 MapReduce job and task scheduling in a virtualized heterogeneous environment

MapReduce scheduler determines how to distribute map/reduce tasks of a job across a cluster of nodes to execute. A MapReduce job can have any number of map and reduce tasks depending upon the requirements of parallelism. Once a job is selected to launch, map and reduce are distributed to different VMs and assigned with a container. Resource requirements of the map and reduce tasks are not the same as map

involves with much disk activity while reduce is more of computing and network activity. Running these tasks in a VM residing in a heterogeneous environment leads to varying job latency. Moreover, co-located VM's interference causes temporary unavailability of shared IO (disk, network) resources. Therefore, it is essential to launch tasks on the right VM based on its heterogeneous performance and resource availability. Key challenges here are to improve job latency and resource utilization. Co-located VM's interference plays a vital role in minimizing the performance [30]-[33] of data-intensive applications in a virtualized environment. Therefore, heterogeneous performance leads to varying job latency and resource under-utilization. Classical MapReduce scheduler is not designed to exploit heterogeneous performance. Most of the Hadoop modified versions for the cloud platform from Hortornworks and MapR perform well only in homogeneous environment. Therefore, it is essential to consider heterogeneous performance to improve job latency and resource utilization, thereby minimizing the service cost.

MapReduce job/task scheduling plays a vital role in improving the performance of the application by satisfying user-defined constraints. There are various scheduling approaches targeting different QoS parameters such as cost [6], latency [19], [20], makespan [21], resource utilization [22], etc. It is always hard to find a solution to optimize all these parameters together. However, schedulers are always based on one or more QoS parameters. MapReduce job scheduling is not exceptional and has seen different job and task schedulers over a decade. MapReduce distribution comes with three basic schedulers for job scheduling: First Come First Serve (FCFS) scheduler, fair scheduler [22], and capacity scheduler [23]. FCFS dedicates the entire cluster resources for a job one after the other as it arrives. Only after the first job gets completed, the next job is executed. Fair scheduler equally shares the cluster resources among a set of jobs. Therefore, every job in a batch has an equal share in a given time. Capacity scheduler reserves the amount of resources for any job.

Task scheduling largely affects the makespan and resource utilization of a system. Makespan is minimized by forming two different queues (IO bound and CPU bound) to classify a batch of heterogeneous MapReduce jobs in [24]. Then, system resource con-

sumption is dynamically monitored at runtime to classify the heterogeneous MapReduce jobs to schedule tasks. Authors claim that the makespan is improved over 20% compared to the classical schedulers. In a virtual environment with heterogeneous capacities, containers for heterogeneous jobs are dynamically decided at runtime by Dazhao Cheng *et al.* in [25] and improved latency and resource utilization by 20%, and 15% respectively. Authors proposed self-adaptive task tuning, where similar hardware configurations are grouped from heterogeneous clusters into several pools. Then, an algorithm continuously updates the list of machines in the pool based on task performance. Genetic algorithm is used to escape from a local optimum for selecting the best configurations.

Two classes of algorithms are proposed in [26] to minimize makespan and total completion time for an offline batch of workloads in a virtual environment. Authors ordered the jobs in a batch under given slot configuration mentioned at the time of submission. During the job execution, the configuration parameters are adjusted in order to consume adequate resources if available. Similar work has been done in [27] to optimize the slot configuration parameters at runtime by learning from previously completed workloads. Authors assume that same batch of jobs are periodically executed on the dataset and claim significant improvement in makespan and resource utilization at runtime. Ming-Chang Lee *et al.* presented a scheduler, JoSS [28], to schedule map/reduce tasks by classifying the job types to design scheduling policy at runtime. Authors classify MapReduce jobs based on job scale, and job type to design scheduling policy to improve data locality for map tasks and task assignment speed. A random forest approach is attempted to predict the optimal slot configuration for different jobs in [29]. Authors use genetic algorithm to explore the configuration parameter solution space to find an optimal combination of parameters. Thus, makespan is improved up to 2.1 times compared to classical MapReduce schedulers.

Performance interference of co-located VMs [34] is predicted for the network, disk, and CPU to achieve efficient scheduling. Authors have designed a prediction-based scheduler to understand interference. Hierarchical clustering is applied in [35] to group cluster hardware based on the performance of tasks (CPU and IO bound) dynamically in

a heterogeneous cluster. IO access prediction of a node is proposed in [36] to optimize MapReduce output writing continuously in a virtualized environment. Authors applied a Markov model for predicting an IO access pattern of a node writing MapReduce VM results and other non-MapReduce VM outputs. By predicting MapReduce output generation to write on the disks, algorithm coordinates the writing of MapReduce outputs continuously to read efficiently later.

Varying resource requirements of tasks during their lifetime complicates job schedulers to fruitfully use the free resources to minimize the job latency, eventually to achieve throughput. To address this challenge, [37] introduces a resource-aware MapReduce scheduler, which breaks the execution of tasks into phases: processing, storage, and data transfer (network). According to this, the phase is a gap between any IO CPU resource access, which takes some time. For instance, when a task involves IO, its CPU can be used by some other task. Therefore, the author focuses on scheduling and allocation at the phase level to avoid resource contention due to too many simultaneous tasks on a machine. Adaptive Task Allocation Scheduler (ATAS) attempts to improve LATE schedulers on a heterogeneous cloud computing environment in [38]. ATAS employs a method to calculate the response time and inspect backup tasks affecting latency and enhance the backup task success ratio. A fine-grained dynamic MapReduce scheduling algorithm is proposed in [40], which significantly minimizes task latency and improves resource utilization. It tracks historical and real-time information obtained from each node to find slow nodes dynamically. In order to further improve cluster performance, it classifies map nodes into high-performing nodes and low-performing nodes for allocating tasks by inferring task profile.

Feng Yan *et al.* [41] proposed a mechanism for task placement in MapReduce considering heterogeneity that exists in processor cores to improve latency. Authors classify cores as fast/slow, and identify MapReduce tasks that require more processing power and assign them to appropriate cores to improve latency. This specifically targets workflows that are completion time sensitive. MapReduce latency is improved with a machine learning algorithm in [42] on a heterogeneous cloud. Authors employed three main aspects: 1. Building a model that can learn system performance by analyzing

historical job information in a cluster. They obtain a list of tasks that have already been executed in every node, task running time, the size of the data block, and other statistical information. 2. Then, the performance value of each node is calculated in the cloud cluster. 3. Finally, based on the performance value, reduce tasks are assigned. ARIA [43] (Automatic Resource Inference and Allocation) develops a Soft Level Objective (SLO), which uses the earliest deadline first job ordering and calculates resource requirements to satisfy job deadlines. Initially, a job profile that comprises the details of mappers, shuffle, sort, and reducers statistics is built based on the previous execution of production workloads. Then, a performance model is constructed for new jobs processing new datasets based on the job profile. Finally, a job is selected that meets SLO with minimal resources, and it is launched. Zaharia *et al.* introduced delay scheduling in [45] to achieve data locality for map tasks to minimize latency. A map task is made to wait for a short time when it did not have the opportunity to achieve data locality. However, to avoid starvation, after a given time has passed, the required data block is moved to another machine and executed non-locally. Tian *et al.* [46] devised a workload prediction on-the-fly to categorize MapReduce workloads into different patterns based on the utilization of computing resources. Authors devised a scheduler with three queues (CPU bound, IO bound, and wait) for a heterogeneous environment to optimize CPU and disk IO resource usage.

2.1.2 Scheduling reduce tasks based on its input size

In general, the output of map tasks is evenly distributed to reduce tasks to balance reduce input in [47]. Authors have introduced an intermediate task between map and reduce phase that balances the load across reduce nodes. FP-Hadoop [9] applies more parallelism in reduce phase by efficiently tackling the problem of reduce input size. FP-Hadoop introduces a new phase, where blocks of intermediate values are processed by intermediate reduce workers in parallel. With this approach, even when all intermediate values are associated with the same key, the central part of the reducing work can be performed in parallel taking the benefit of the computing power of all available workers. A set of sample map tasks is used to track its output in [10]. Based on the output, it evenly divides the key range space and directs subsequent map tasks output to

split in the same fashion to load balance among reduce tasks. Authors consider cluster heterogeneity while allocating reduce tasks by finding the amount of data processed per unit of time dynamically. To mitigate partitioning skew, rather than dividing partitions to balance the load among reduce tasks, the amount of resources allocated to reduce task is increased/decreased in [48]. Therefore, each reduce task gets different size of resources based on the size of its input; consequently, the completion time of a job is minimized.

Therefore, each reduce task gets different size of resources based on the size of its input; consequently, the completion time of a job is minimized. However, in a heterogeneous environment, despite allocating more resources for a reduce task, if the performance of a node is poor, then there is no guarantee in minimizing the completion time. The uneven distribution of map outputs to the reduce tasks are discussed in [49]. It considers historical records for constructing profiles for every job type because production jobs are routinely launched in production clusters. It dynamically calculates the size of the partition for each reduce task and allocates adequate resources to reduce tasks. For instance, reduce task having more input will get more resources than other reduce tasks. Reduce task is compute-intensive as it runs the major part of algorithms. Therefore, it dynamically adjusts the size of the container (CPU and memory) to reduce tasks according to the partition size. Gufler's *et al.* [50] worked on the partitioning problem in scientific applications to handle the uneven size of data distribution of mapper outputs. Authors have focused on minimizing reduce task execution time, balancing the load evenly among reduce tasks. Fan Yuanquan *et al.* used support vector machine to predict the performance of target node to choose the right one for reduce task and heterogeneity-aware partitioning that balances the skewed reduce input size in [51].

2.1.3 Minimizing the size of intermediate data during the shuffle phase in a virtual environment

Shuffle phase in MapReduce execution sequence consumes huge network bandwidth taking a major portion of job latency. A research finding [8] shows that 26%-70% of job latency is due to the shuffle phase. It sets a trade-off between job latency and network bandwidth. To minimize job latency, assign more bandwidth, which leads to

pay more, and vice versa. At times, even though having allocated more bandwidth, due to co-located VM's dynamic bandwidth consumption behavior, there is a chance to face network bottleneck. Miguel Liroz *et al.* introduced a new phase, intermediate reduce in [9], to minimize the map output further. It sits in between combiner and reducer execution. Authors achieved up to 10 times reduction in reduce phase latency, and 5 times reduction in job latency. A small fraction of intermediate output is used to determine which reduce task can get more input, and load is balanced using a distributed method in [10]. This work also considers the heterogeneity of computing resources for more precise load balancing.

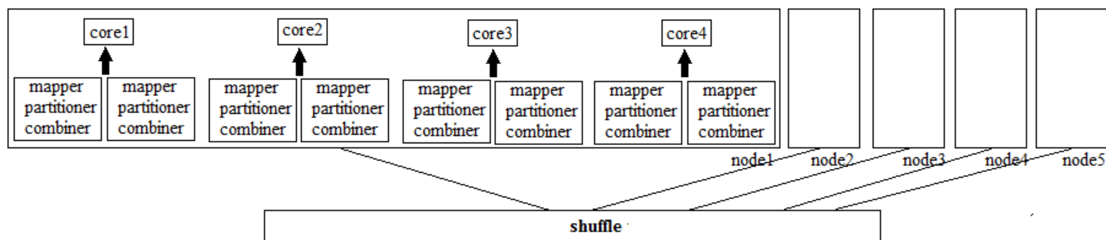


Figure 2.1 Default combiner

Rather than balancing the load of reduce task input, there is a chance to minimize the size of intermediate output to minimize the job latency and network bandwidth consumption. Default combiner [4] minimizes the number of intermediate records transferred over the network, as shown in Figure 2.1. Default combiner is executed an arbitrary number of times on the map output records before spilling them into a disk. It runs in parallel with map task and shrinks the size of data stored into a disk. Consequently, it minimizes the amount of data spilled into a disk, so disk access is largely minimized and also the amount of data in the shuffle phase. However, the downside of default combiner is, until combiner function finishes its execution, map function does not end to enter the shuffle phase. Moreover, the number of times the combiner function executed is not fixed. Therefore, a map task finishes its execution and sends intermediate results to reduce nodes at a different time.

A "Per Node Combiner" (PNC) [11] was proposed by Lee *et al.* to minimize the amount of shuffle data, thereby minimizing job latency. Unlike running a dedicated

combiner function (Figure 2.2) for each map task, a single combiner is executed in each node. All map tasks running in a specific node writes their intermediate output in a common distributed cache database (Redis) rather than writing in an in-memory buffer. The combiner function is executed when the size of the distributed cache fills up to a specific threshold, and the final result is sent over the network only when the last map task in the specific node arrives. However, one cannot easily determine which map task could be the last map task in each node if a batch of jobs is running. iShuffle is proposed in [12] by Yanfei Guo *et al.* to perform the shuffle-on-write operation and move to reduce nodes by decoupling shuffle and reduce task execution. So, shuffle is allowed to make decisions independently by predicting the partition size dynamically to balance the reduce inputs. It achieved 30.2% improvement in overall job latency. IO overhead in JVM also affects shuffle time, and it is addressed by Wang *et al.* in [13]. By default, Hadoop framework uses java stack based IO protocols (HttpServlets) for shuffling intermediate data. It performs 40%-60% slower when compared to IO framework written in C. Therefore, authors proposed a JVM bypass shuffling to eliminate this overhead by leveraging TCP/IP and remote direct memory access to speed up shuffle phase. JBS achieves up to 66.3% reduction in job latency and lowers the CPU time by 48.1%.

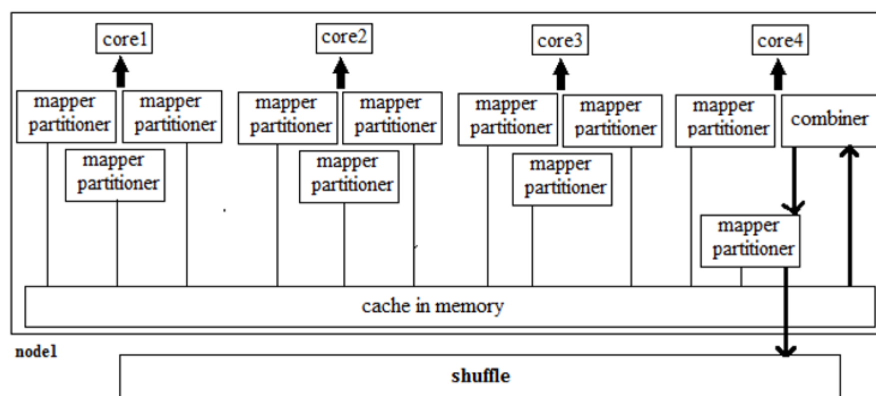


Figure 2.2 Per Node Combiner (PNC) [11]

Huan Ke *et al.* [14] proposed three approaches to minimize network traffic during the shuffle phase: intra-machine aggregation, inter-machine aggregation, and in-cloud aggregation. Authors developed an aggregator service that can be launched anywhere in the cluster independent to reduce tasks for decreasing map output size before sending

to reduce tasks. In the first approach, authors launch aggregator whenever a set of map tasks in a node tends to produce huge intermediate records. It merges all intermediate records generated by all map tasks in the same node before sending over the network. In inter-machine data aggregation, the aggregator is launched in any one of the nodes running map tasks. Other map task nodes send its intermediate results to the node running aggregator service. Authors try to minimize the number of intermediate records at rack level before sending to reduce task nodes. In cloud aggregation, aggregator service runs anywhere in the cluster. Nodes running map tasks send their intermediate results to this node, which decreases the number of records and forwards it to the reduce nodes.

Push-based aggregation and parallelizing shuffle phase are introduced in [15] to minimize network traffic and efficiently use available network bandwidth in data-centers. Authors proposed IRS-based on in-network aggregation tree, and SRS-based shuffle aggregation subgraph based on data-center topology. Authors also designed scalable forwarding schemes based on Bloom filters to implement in-network aggregation over massive concurrent shuffle transfers. Authors saved network traffic by 32.87% on an average for small-scale shuffle and 55.33% over large scale shuffle in data-center. Wei *et al.* [16] aim to minimize the overall network traffic, manage workload balancing, and eliminate network hotspots to improve performance in arbitrary network topologies. Uneven distribution of map output to different nodes causes more traffic at a specific portion of the data-center. An algorithm "smart shuffling" is proposed to suggest a set of candidate nodes to launch reduce tasks. Camdoop [17] solves the shuffling bottleneck due to intermediate records by decreasing the network traffic. Authors perform shuffling using hierarchical aggregation before sending over the network. However, camdoop is only effective in special network topology, such as 3D torus network, and its performance degrades sharply in common network topologies adopted by data-centers.

Liang and Lau [18] introduced bandwidth-aware shuffler to maximize the utilization of network bandwidth, which leads to increase shuffle throughput at the application level. Authors argue that random source selection policy introduces the network bottleneck in case of heterogeneous bandwidth in the cluster while choosing nodes for reduce

tasks. Authors proposed a partial greedy source selection, which sets a load count in each slave node to track how many number of fetches may happen for a shuffle shortly. A node that has the largest load count will be indicated as reduce node, which needs the maximum network bandwidth. It incurs a small scheduling overhead. However, authors claim that the proposed algorithm shortens the reduce phase latency 29% and overall job latency up to 21%.

2.1.4 Block placement schemes in HDFS

Sometimes, block placement also determines the latency of a job in a heterogeneous virtualized environment. Xie *et al.* consider the capacity of nodes to distribute data in a heterogeneous environment to improve the performance of MapReduce applications in [44]. Authors introduce a novel file system for data distribution that relocates data during execution to improve performance. This proposed algorithm has different functionalities: break input data based on the capacity of machines, redistribute data blocks based on current CPU processing speed, and fresh incoming data handling. This work achieves a 33% improvement in minimizing latency by balancing the workloads in different VMs on a heterogeneous environment. A replica balanced distribution tree structure is designed to achieve optimal data blocks placement policy in [52]. Authors focus on minimizing the global data access cost and the number of non-local execution and achieved up to 32.5% improvement over classical MapReduce schedulers.

A novel data block distribution technique is proposed by Vrushali Ubarhande *et al.* in [53] for cloud heterogeneous environment to improve makespan. In this work, a speed analyzer is used to find the computing performance of virtual nodes to distribute blocks. Similarly, Chia-Wei Lee *et al.* proposed a dynamic data block placement scheme [54] to minimize the number of non-local execution based on the virtual node's computing capacity in a heterogeneous virtual environment. Authors indicate that virtual node's processing capacity is not the same for different types of MapReduce jobs. Therefore, the data blocks of each workload are placed based on the computing capacity for the respective workload. Consequently, authors improved performance over 23.5% compared to the classical MapReduce schedulers. MRA++ [39], a data block placement technique, is introduced for a heterogeneous environment. Few map tasks

are used as training tasks to explore the heterogeneous performance and capacity before distributing blocks. Typically, slower nodes are not preferred as it will lead tasks to be stragglers. Therefore, authors employ a classification method to group virtual nodes based on the computing capabilities using the information collected during training time. This minimized the job latency and balanced the load across nodes.

2.1.5 Bin packing tasks

Utilizing maximum resources of VMs for heterogeneous workloads is a challenging task. Bin packing [56], [60] tasks improves resource utilization, and it has a wide variety of applications. Task consolidation using bin packing with meta-heuristic algorithms [55], [57], [61], [66], [67] is widely applied. When the number of constraints increases, the possible solution space decreases and finding an optimal solution by avoiding unfavorable solutions from solution space becomes complex. Some works focus on heterogeneous bin capacity [55], [58], [59], [62], [64], [65] while some works focus on varying size of workloads [63], [68] to pack them into homogeneous/heterogeneous bins. Despite there is no known application of bin packing of map/reduce tasks, we just mentioned the bin packing problems with heterogeneous workloads and bin capacities in different application areas.

2.2 Key Observations

2.2.1 MapReduce job and task scheduling in a virtualized heterogeneous environment

Map/reduce tasks are scheduled based on either performance-aware or resource-aware or interference-aware techniques.

- Performance-aware task scheduling considers the performance of nodes based on the past task execution.
- Resource-aware scheduling focuses on scheduling tasks based on the availability of resources.
- Interference-aware task scheduling predicts the disturbance of co-located VMs and schedules tasks accordingly.

Merely understanding the performance of a node from the past epoch may not improve task latency and resource utilization. For example, the map phase requires more of disk IO and CPU, while reduce phase requires network IO and CPU. When performance for a node is calculated based on CPU and Disk IO, reduce task might face bottleneck due to network congestion. Performance of a VM also varies dynamically due to the interference of co-located VMs. Also, independent jobs go for non-local execution due to static scheduling decision. Moreover,

- all these works suffer from computational load imbalance as data locality is mandatory to minimize the job latency.
- data locality is significantly affected while concentrating on the performance of a node to place tasks. .
- dynamically tuning container configurations minimizes the latency but at the cost of resource under-utilization.

2.2.2 Scheduling reduce tasks based on its input size

Unlike map tasks receiving same size of input, reduce tasks receive various size of input. Existing works either balance the input size for all reduce tasks or place them based on the computing power of each node. Authors do not consider the dynamic performance of VMs.

2.2.3 Minimizing the size of intermediate data during the shuffle phase

It is important to minimize the number of intermediate records transferred in the shuffle phase rather than supplying more network bandwidth that results in increased service cost. Interestingly, PNC [11] performs node level intermediate records aggregation in the memory itself. All map tasks of a specific job writes its intermediate data in the in-memory buffer. Once the buffer exceeds a threshold, PNC is applied on intermediate records and the results are moved to reduce nodes. PNC involves a several shuffle emits during the job life cycle. Moreover, once a first shuffle emit happens, all the reduce tasks of a job should be launched to collect the intermediate records as there is no

spilling in PNC. Holding containers for reduce tasks until job completion may clog the available resources in the virtual cluster leading to less throughput. While this method promises a reduction in the number of records in shuffle phase, there is still a possibility to minimize the number of intermediate records further.

2.2.4 Block placement schemes in HDFS

Heterogeneous VM capacities are not considered while placing the data blocks to minimize the number of non-local execution, which in turn minimizes job latency. Learning the history of workloads and predicting the block placement for future workloads may not be meaningful if the jobs are not homogeneous.

2.2.5 Bin packing tasks

While placing heterogenous tasks in heterogeneous VMs, a large portion of virtual cluster resource is wasted. So, finding the right combination of tasks to schedule in each VM is a possible option for task scheduling. To the best of our knowledge, bin packing has not been applied for MapReduce task scheduling.

2.3 Problem Definition

Tuning MapReduce job and the task scheduler to improve job latency, makespan, and resource utilization by exploiting underlying heterogeneities in the cloud environment.

2.4 Research Objectives and Works

Considering the outcomes of literature survey, we proposed a set of methods for MapReduce task and job scheduler to improve job latency, makespan, and resource utilization, as given below.

- Objective 1: Scheduling map/reduce tasks to improve job latency and resource utilization. For this, we proposed

    1. Dynamic Ranking based MapReduce Job Scheduler (DRMJS) to exploit heterogeneous performance.
    2. Multi-Level Per Node Combiner (MLPNC) to minimize the number of intermediate records in the shuffle phase.

3. Reduce task scheduling based on performance rank after MLPNC.

- Objective 2: Scheduling MapReduce jobs to improve makespan and resource utilization. For this, we proposed

    1. Roulette Wheel Scheme (RWS) based data block placement in HDFS to minimize job latency.
    2. Constrained 2-dimensional bin packing map/reduce tasks using Ant Colony Optimization (ACO) to exploit heterogeneous VM capacities and workloads.
    3. Fine-Grained Data-Locality Aware (FGDLA) job scheduling to minimize the number of intermediate records for a batch of jobs.

Chapter 3

MapReduce Task Scheduling

3.1 Proposed Methodologies

Resource requirements of map/reduce tasks, and heterogeneous performance of Hadoop VMs pose a major challenge for MapReduce task schedulers to improve job latency and resource utilization in a virtualized environment. Therefore, we proposed the following methods in order to improve the performance of MapReduce task scheduler in a virtualized environment.

1. Dynamic Ranking based MapReduce Job Scheduler (DRMJS) to exploit heterogeneous performance

2. Multi-Level Per Node Combiner (MLPNC) to minimize the number of intermediate records in the shuffle phase.

3. Reduce task scheduling based on performance rank after MLPNC.

Firstly, DRMJS is proposed to improve MapReduce job latency and resource utilization by exploiting heterogeneous performance. The DRMJS calculates the performance score for each Hadoop virtual machine based on CPU and Disk IO for map tasks, CPU and Network IO for reduce tasks separately. Then, a rank list is prepared for scheduling map tasks based on map performance score, and reduce tasks based on reduce performance score. Ultimately, DRMJS improved overall job latency, makespan, and resource utilization up to 30%, 28%, and 60%, respectively, on average compared to existing MapReduce schedulers. To improve job latency further, MLPNC is introduced to minimize the number of intermediate records in the shuffle phase, which is responsible for

the significant portion of MapReduce job latency. In general, each map task runs a dedicated combiner function to minimize the number of intermediate records. In MLPNC, we split the combiner function from map task and run a single MLPNC in every Hadoop virtual machine for a set of map tasks of the same job. These map tasks write its output to the common MLPNC, which minimizes the number of intermediate records level by level. Ultimately, MLPNC improved job latency up to 33% compared to existing MapReduce schedulers for a single job. These methods are discussed in detail in the subsequent sections.

3.1.1 Dynamic Ranking based MapReduce Job Scheduler (DRMJS)

Performance of VMs is highly dynamic due to different types of hardware and co-located VM's resource consumption behavior. It is beneficial allocating map and reduce tasks based on the heterogeneous performance of each VMs. In order to calculate the performance of VMs dynamically, we need to develop a model that captures the resource usage of each VM periodically. Consider s Hadoop VMs hosted on t PMs. CPU performance of jth VM in i^{th} PM_{ij}^{CPU} is calculated by finding the PM having maximum CPU frequency (CPU_freq) among t PMs in which Hadoop VMs have been hosted, as given in Equation 3.1.

$$VM_{ij}^{CPU} = \frac{VM_{ij}^{CPU_freq}}{max(\forall_i, PM_i^{CPU_freq})} \tag{3.1}$$

Note that the performance of all VMs hosted in a PM is not necessarily to be the same. We observed that many VMs hosted in a PM may have storage allocated in different Hard Disk Drives (HDD), data transfer on different Network Interface Card (NIC), and executed by different cores, as shown in Figure 3.1. For instance, the contention of disk IO may be different in different HDD. Therefore, we calculate the performance of all VMs hosted in a PM. Disk IO performance of jth VM in i^{th} PM (VM_{ij}^{DiskIO}) is calculated using Equation 3.2 based on the current disk bandwidth rate of jth VM in i^{th} PM ($VM_{ij}^{curr_disk_band}$) over the disk bandwidth of kth disk in i^{th} PM ($PM_{ik}^{Disk_band}$). Network IO performance of jth VM in i^{th} PM (VM_{ij}^{NetIO}) is calculated using Equation 3.3 based on the current bandwidth rate of jth VM in i^{th} PM ($VM_{ij}^{curr_net_band}$) over the Nework bandwidth of lth NIC in i^{th} PM ($PM_{il}^{Net_band}$).

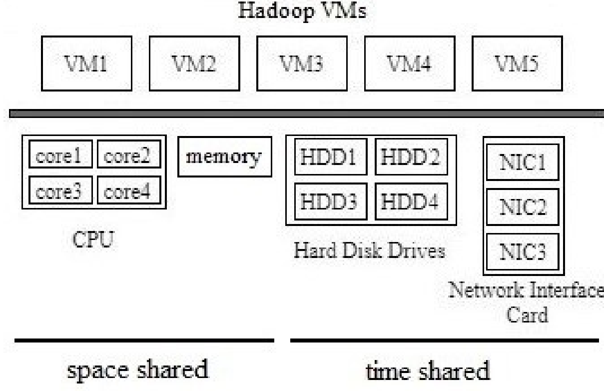


Figure 3.1 VMs sharing resources in a PM

$$\forall_{i,j}, \quad VM_{ij}^{DiskIO} = \forall_k, \frac{\sum VM_{ij}^{curr_disk_band}}{PM_{ik}^{Disk_band}} \tag{3.2}$$

$$\forall_{i,j}, \quad VM_{ij}^{NetIO} = \forall_l, \frac{\sum VM_{ij}^{curr_net_band}}{PM_{il}^{Net_band}} \tag{3.3}$$

Resource requirements of map and reduce tasks are different as map task requires CPU and disk activities while reduce task demands more of CPU and network activities. Therefore, we need to place map/reduce tasks based on the map/reduce node performance rather than the overall performance of a VM. Before calculating the performance of a VM for map and reduce tasks separately, we calculate the influence of jth VM in i^{th} PM for map ($VM_{ij}^{map_inf}$) and reduce ($VM_{ij}^{reduce_inf}$) by considering the latency of last z map/reduce tasks executed in j^{th} VM using Equation 3.4 and Equation 3.5).

$$\forall_j, \quad VM_{ij}^{map_inf} = min\left(\forall_z, \frac{map_latency_{jz}}{\sum_{m=1}^{z} map_latency_{jm}}\right) \tag{3.4}$$

$$\forall_j, \quad VM_{ij}^{reduce_inf} = min\left(\forall_z, \frac{reduce_latency_{jz}}{\sum_{m=1}^{z} reduce_latency_{jm}}\right) \tag{3.5}$$

Existing approaches calculate the overall performance of a VM regardless of type of tasks. But, in DRMJS, we calculate the performance of VM for map and reduce separately. Therefore, map performance ($VM_{ij}^{map_perf}$) is calculated based on CPU and Disk IO of respective VM in each PM using Equation 3.6). Similarly, reduce performance ($VM_{ij}^{reduce_perf}$) is calculated using CPU, and Network IO of respective VM in each PM using Equation 3.7).

$$\forall_{i,j}, VM_{ij}^{map_perf} = VM_{ij}^{CPU} \times (1 - VM_{ij}^{DiskIO}) \times (1 - VM_{ij}^{map_inf}) \tag{3.6}$$

$$\forall_{i,j}, VM_{ij}^{reduce_perf} = VM_{ij}^{CPU} \times (1 - VM_{ij}^{NetIO}) \times (1 - VM_{ij}^{reduce_inf}) \tag{3.7}$$

Map and reduce performance of all Hadoop VMs are sorted using Equation 3.8 and Equation 3.9 to find the best nodes for map tasks (*map_rank*) and redue tasks (*reduce_rank*) separately.

$$map_rank = sort(VM_{ij}^{map_perf}) \tag{3.8}$$

$$reduce_rank = sort(VM_{ij}^{reduce_perf}) \tag{3.9}$$

There can be more than one VM having the same score, which gives the option to choose any one of the VM for allocating tasks. In short, Algorithm 1 summarizes the dynamic rank calculation for heterogeneous performance of virtual nodes.

Algorithm 1: Dynamic Performance Rank Calculation

1 **Notation:**
2 PM_i- i^{th} PM in CDC
3 $VM_{ij}-$ j^{th} VM in i^{th} PM in Hadoop virtual cluster
4 **Input:**
5 VM parameters (virtual CPU, virtual Disk, virtual Network)
6 System parameters (CPU, Disk, Network)
7 **Output:**
8 performance rank of map and reduce tasks separately for each VM
9 **while do**
10 calculate the CPU performance VM_{ij}^{CPU}
11 calculate the Disk IO rate VM_{ij}^{DiskIO} (Disk IO includes both HDFS and LFS IO)
12 calculate the network performance VM_{ij}^{NetIO}
13 calculate the map task influence $VM_{ij}^{map_inf}$
14 calculate the map task performance $VM_{ij}^{map_perf}$
15 sort $VM_{ij}^{map_perf}$ in descending order to find *map_rank*
16 calculate the reduce task influence $VM_{ij}^{reduce_inf}$
17 calculate the reduce task performance $VM_{ij}^{reduce_perf}$
18 sort $VM_{ij}^{reduce_perf}$ in descending order to find *reduce_rank*
19 **end**

3.1.2 Map and reduce task scheduling based on performance rank

Algorithm 2 explains how map tasks of nth job (J_n) are scheduled by getting map rank list. map_{np} denotes p^{th} map task of nth job and initlized to 0 as map tasks are not yet

Algorithm 2: Heterogeneous performance aware map task scheduling

1. **Input:** map tasks of a job, map rank of VMs
2. **Output:** assign map tasks to the right VM_{ij}
3. m $-$ number of map tasks of a job J_n
4. $map_{np}-$ p^{th} map task of nth job
5. $\forall_p, map_{np} = 0$
6. $C_n = 0-$ number of completed map tasks of job J_n
7. **while** $C_n < m$ **do**
8.   Pick up a map task (p) from task list
9.   **if** map_{np} == *0* **then**
10.   Choose top 10% VMs from $VM_{ij}^{map_perf}$ rank list
11.   **while** *until 10% nodes* **do**
12.   **if** *containers possible for* map_{np} && *data locality* **then**
13. map_{np}=1
14.   Launch map task, C_n++
15.   break
16.   **end**
17.   **end**
18.   **end**
19.   **else if** map_{np} == *0* **then**
20.   Choose rest 90% VMs from $VM_{ij}^{map_perf}$ rank list
21.   **while** *until 90% nodes* **do**
22.   **if** *containers possible for* map_{np} && *data locality* **then**
23. map_{np}=1
24.   Launch map task, C_n++
25.   break
26.   **end**
27.   **end**
28.   **end**
29.   **else if** map_{np} == *0* **then**
30.   perform non-local execution
31. map_{np}=1
32.   Launch map task, C_n++
33.   **end**
34.   **else**
35.   add map_{np} into task queue
36.   **end**
37. **end**

scheduled. Initially, top 10% of map nodes from the rank list is chosen to schedule map tasks. If container for map task is possible, then map task is scheduled and map_{np} is assigned to 1. If there are not enough resources available to form container and no

Algorithm 3: Heterogeneous performance aware reduce task scheduling

```
1  Input:    reduce tasks of a job, reduce rank of VMs
2  Output:   assign reduce tasks to the right VM_ij
3  r − number of reduce tasks of a job J_n
4  reduce_nq − q^th reduce task of n^th job
5  ∀_q, reduce_nq = 0
6  C_n = 0 − number of completed reduce tasks of job J_n
7  while C_n < r do
8      Pick up a reduce task (q) from task list
9      if reduce_nq == 0 then
10         Choose top 10% VMs from VM_ij^{reduce_perf} rank list
11         while until 10% nodes do
12             if containers possible for reduce_nq then
13                 reduce_np=1
14                 Launch reduce task, C_n++
15                 break
16             end
17         end
18     end
19     else if reduce_nq == 0 then
20         Choose rest 90% VMs from VM_ij^{reduce_perf} rank list
21         while until 90% nodes do
22             if containers possible for reduce_nq then
23                 reduce_nq=1
24                 Launch reduce task, C_n++
25                 break
26             end
27         end
28     end
29     else
30         add reduce_nq into task queue
31     end
32 end
```

data locality is possible, then rest 90% of the nodes in the rank list is considered for scheduling map tasks. If there is no data locality possible in any of the nodes in the rank list, then non-local execution is performed. If there is no scope for a map task at this moment, it is added back to the task queue. MRAppMaster usually checks for data-locality by default. Similarly, Algorithm 3 schedules reduce tasks on the top 10% high performing VMs. If it is not possible to form containers, then rest 90% of nodes

are inspected for containers to launch reduce tasks. If more than one VM has the same rank, then VM, which has significant network bandwidth, is chosen to launch containers for reduce tasks.

3.1.3 Scheduling reduce tasks based on its input size

Even though performance rank helps in finding VM that is capable of minimizing reduce task latency, as given in Algorithm 3, still there is a chance to improve reduce task latency further. Unlike map tasks receiving same size of input, the input size of reduce tasks is not uniform and unpredictable. Therefore, scheduling reduce tasks without concerning its input size could increase job latency. For instance, consider a reduce task *r* taking huge input and gets allocated with a low performing reduce node for execution. Consider another reduce task *r+1* taking less input and gets allocated with high performing reduce node for execution. *r* takes more time to complete as it is being executed by low performing reduce node. But, assigning *r* to high perfoming node helps to minimize task latency, consequently, overall job latency is also improved. To find the input size of each redue task before map tasks completed, we dynamically interact with partitioner to track the amount of each reduce task's input size. With this information, we allocate reduce tasks that get huge input onto high performing node using performance rank.

In order to assign reduce tasks to the right high performing reduce node, we initially classify all VMs into any one of four classes (C_1, C_2, C_3, and C_4) using values of ($VM_{ij}^{reduce_perf}$). Table 3.1 contains values of all components, as in Equation 3.7. We categorize the values of ($VM_{ij}^{reduce_perf}$) into five (low, below average, average, above average, and high) to set the ranges for C_1, C_2, C_3, and C_4. Values filled up for the components in Table 3.1 are based on the observation from our experiment, so as to pick the right VM for reduce tasks. The class ranges are formed as follows: C_1 = 1 to 0.343, C_2 = 0.342 to 0.064, C_3= 0.063 to 0.008, C_4 = 0.007 to 0.001. Once rank list for reduce is prepared, each node with its reduce performance score is compared with the ranges of C_1, C_2, C_3, and C_4. We considered only four classes as forming ranges would be too narrow for cases below average. As given in Algorithm 4, partition size of respective reduce task from all map tasks is aggregated globally using Equation 3.10.

Table 3.1 Performance class

Category	VM_{ij}^{CPU}	VM_{ij}^{NetIO}	VM_{ij}^{reduce}	$VM_{ij}^{reduce_perf}$
low	0.1	0.1	0.1	0.001
below average	0.2	0.2	0.2	0.008
average	0.4	0.4	0.4	0.064
above average	0.7	0.7	0.7	0.343
high	1	1	1	1

Then, we find the maximum partition size, using which the probability of each partition is calculated with Equation 3.11 to assign reduce task to the node that belongs to any one of the classes (C_1, C_2, C_3, and C_4).

It is important because, relatively equal performance nodes are put into the same class. So, even if there is no container possible for reduce task in a node that belongs to C_1, another node in the same class is inspected for containers. If none of the nodes in the same class is possible to form container, then immediate next class (C_2) is inspected if particular reduce task waits up to 30 seconds to avoid starving. Because, all reduce tasks should be launched to receive inputs from map tasks. If nodes in C_2 also don't have sufficient resources, then reduce task is added back into task queue. Because, assigning reduce tasks (intended for C_1) in C_3 or C_4 may lead to huge latency. Similarly, reduce tasks that fall on C_2 may be tried with C_3, if there is no possibility of containers in C_2. If there are no resources in C_2, and C_3, then immediate upper class C_1 can be tried if there are no other tasks intended to launch in C_1. The same way, reduce tasks that fall in C_3 may be tried in other classes in a sequence (C_4/C_2/C_1) ensuring there are no other reduce tasks intended in C_2 and C_1. Finally, reduce tasks that belong to C_4 can be assigned with any of the upper classes (inspected in C_3/C_2/C_1 sequence) if there are no resources available in C_4.

Algorithm 4: Reduce task scheduling based on its input size and performance

---

Notation:

m$-$ the number of map tasks of job J_n

$partition_size_{nq}$ $-$ partition size of qth reduce task of nth job

$VM_P_p^q$ − partition for q^{th} reduce task from p^{th} map task
$reduce_j^{nq}$ − assign q^{th} reduce task of nth job in j^{th} VM
$probability_{nq}$ − q^{th} reduce task partition of nth job
r − number of reduce tasks of J_n
$\forall q,\, reduce_j^{nq} = 0$ // q^{th} reduce task of nth job
$C_n = 0$ // number of completed reduce tasks of job J_n
Input: partition size for each reduce task and VMs performance
Output: assign reduce tasks onto the right VM
calculate partition size of each reduce task from all map nodes
$partition_size_{nq}$=$\sum_{p=1}^{m} size(VM_P_p^q)$ (3.10)
$max = max(partition_size_{nq})$
reduce rank is divided into 4 classes based on values given in Table 3.1
$\forall q,\, probability_{nq} = partition_size_{nq}/max$ (3.11)
Pick up a reduce task (q) from the task list
while $reduce_j^{nq}$!=1 && $C_n < r$ **do**
    if $probability_{nq}$ falls in C_1 **then**
        if container_possible
            assign $reduce_j^{nq}$ in C_1
            q++, C_n++, $reduce_j^{nq}$=1
        end
        elseif
            inspect other nodes in C_1
            assign $reduce_j^{nq}$ in C_1
            q++, C_n++, $reduce_j^{nq}$=1
        end
        elseif q is starving for 30 seconds
            inspect other nodes in C_2
            assign $reduce_j^{nq}$ in C_2
            q++, C_n++, $reduce_j^{nq}$=1
        end
        else
            add q into the task queue
    end
    else if $probability_{nq}$ falls in C_2 **then**
        if container_possible
            assign $reduce_j^{nq}$ in C_2
            q++, C_n++, $reduce_j^{nq}$=1
        end

elseif

inspect other nodes in C_2

assign $reduce_j^{nq}$ in C_2

q++, C_n++, $reduce_j^{nq}$=1

end

elseif q is starving for 30 seconds

inspect other nodes in C_3/C_1

assign $reduce_j^{nq}$ in C_3/C_1

q++, C_n++, $reduce_j^{nq}$=1

end

else

add q into the task queue

end

else if $probability_{np}$ falls in C_3 **then**

if container_possible

assign $reduce_j^{nq}$ in C_3

q++, C_n++, $reduce_j^{nq}$=1

end

elseif

inspect other nodes in C_3

assign $reduce_j^{nq}$ in C_3

q++, C_n++, $reduce_j^{nq}$=1

end

elseif q is starving for 30 seconds

inspect other nodes in $C_4/C_2/C_1$

assign $reduce_j^{nq}$ in $C_4/C_2/C_1$

q++, C_n++, $reduce_j^{nq}$=1

end

else

add q into the task queue

end

else

if container_possible

assign $reduce_j^{nq}$ in C_4

q++, C_n++, $reduce_j^{nq}$=1

end

elseif

inspect other nodes in C_4

assign $reduce_j^{nq}$ in C_4

q++, C_n++, $reduce_j^{nq}$=1

end

elseif q is starving for 30 seconds

inspect other nodes in $C_3/C_2/C_1$

assign $reduce_j^{nq}$ in $C_3/C_2/C_1$

q++, C_n++, $reduce_j^{nq}$=1

end

else

add q into the task queue

end

end

Figure 3.2 exhibits the overall workflow of DRMJS. Hadoop virtual cluster is formed by launching Hadoop VMs in various PMs hosted with other general purpose VMs. Consider a set of PMs ($node_1$,...,$node_n$) in different racks. We deploy Hadoop 2.7.0 for our experiment and work with YARN. YARN is a cluster resource management tool and contains two major services: Resource Manager (RM), and Node Manager (NM). RM is a master process that manages cluster resources and schedules YARN applications (MapReduce, Spark, HPC, etc.). NM runs in every Hadoop VMs to carry out the commands delivered by RM. Initially, the user submits MapReduce jobs at RM, which are added then to the job queue. RM launches MapReduce Application Master (MRAppMaster) for each MapReduce job to manage job life cycle, schedule map/reduce tasks, guarantee fault tolerance, etc., DRMJS is run in any one of the Hadoop VMs (preferably in RM) and dynamically collects information such as VM resource usage and PMs resource availability via Heartbeat message to calculate performance score of each VM for map and reduce tasks separately using Algorithm 1. Based on the performance score, rank is prepared. MRAppMaster receives the performance rank list, picks top VMs, and requests RM to launch container in preferred VM for map/reduce tasks using Algorithm 2 and Algorithm 3. Further, based on the size of input, reduce tasks are scheduled using Algorithm 4.

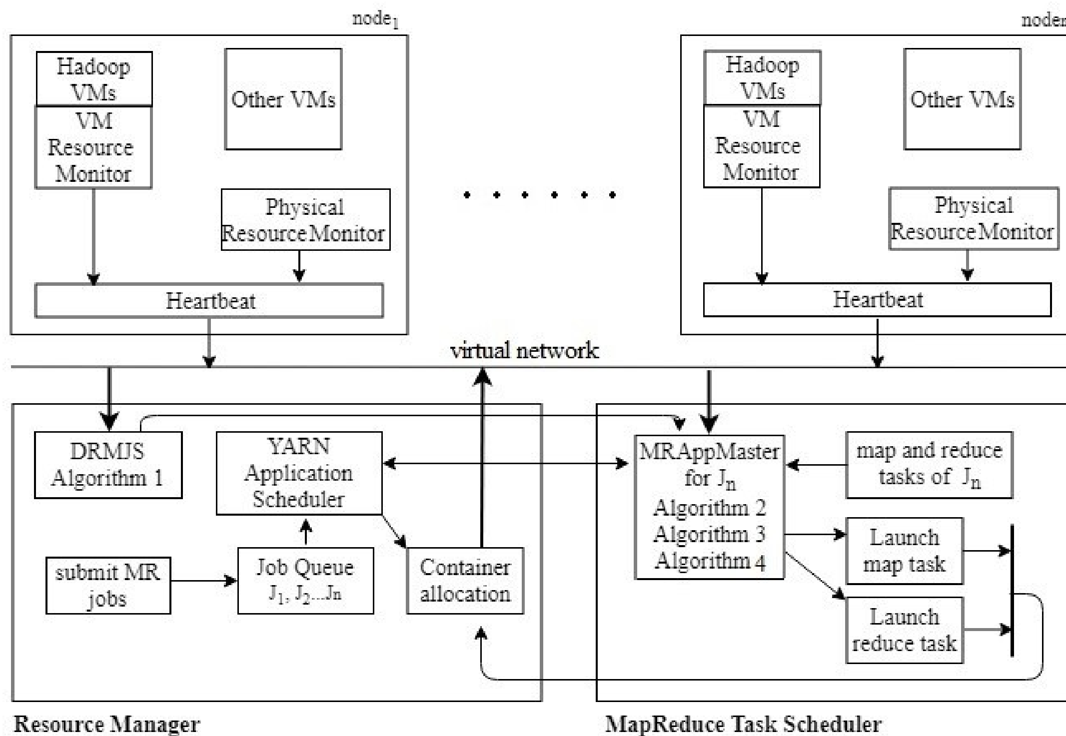


Figure 3.2 Workflow of DRMJS

3.1.4 Multi-Level Per Node Combiner (MLPNC)

To improve job latency further, MLPNC is introduced to minimize the number of intermediate records in the shuffle phase, which is responsible for the significant portion of MapReduce job latency. In general, each map task runs a dedicated combiner function (as shown in Figure 2.1) to minimize the number of intermediate records. PNC [11] is used to minimize the number of intermediate records in the shuffle phase at the node-level (as shown in Figure 2.2) by splitting combiner function from map tasks and running a single combiner for all map tasks per node. The significant constraint of PNC is, once in-memory cache is filled up 80%, all the reduce tasks have to be launched to receive map outputs. It causes all reduce tasks to hold containers (resources) and wait for a long time to receive its whole input. Although it reduces job latency in large proportion, yet there is a possibility to minimize the number of intermediate records further. In MLPNC, we split the combiner function from map task and run a single MLPNC in every Hadoop virtual machine for a set of map tasks of the same job. These map tasks write its output to the common MLPNC, which minimizes the number of

intermediate records level by level until there is no further possibility in minimizing the intermediate records, unlike PNC. Therefore, reduce tasks are launched a little later.

As shown in Figure 3.3, consider node1 containing four cores, in which first three cores are executing three map tasks concurrently, but core4 is running a combiner function level by level. All map tasks write its output into the common cache memory (Memcache), which is used to keep intermediate records in memory itself without spilling into disks. As shown in Figure 3.4, in-memory cache is initialized with S MB that gets filled circular fashion. P is a current pointer in the cache. When cache reaches a threshold T MB ($T \ll S$), two combiners $COMB_1$, $COMB_2$ are launched in separate threads, as given in Algorithm 5 for wordcount job pseudocode. There are three functions in the map task of wordcount job. setup() establishes the connection between the map task (map_id) and Memcache running in the respective node (node_id). map() tokenizes the input value and assigns 1 to each word. cleanup() disconnects map task from the Memcache. Combiner function aggregates all the values of each word and adds up to the result. $COMB_1$ takes 0 to $T/2$ MB, and $COMB_2$ takes $T/2+1$ to T MB of the cache. The output of these combiners and current output of map tasks are written into cache until ($T+1$ to (P mod S)) $<= T$. When (P mod $S == T/2$) is met, $COMB_3$ is launched on the current map tasks output and previous combiner results. In this way, combiner execution is repeated until there is no further reduction in the map output

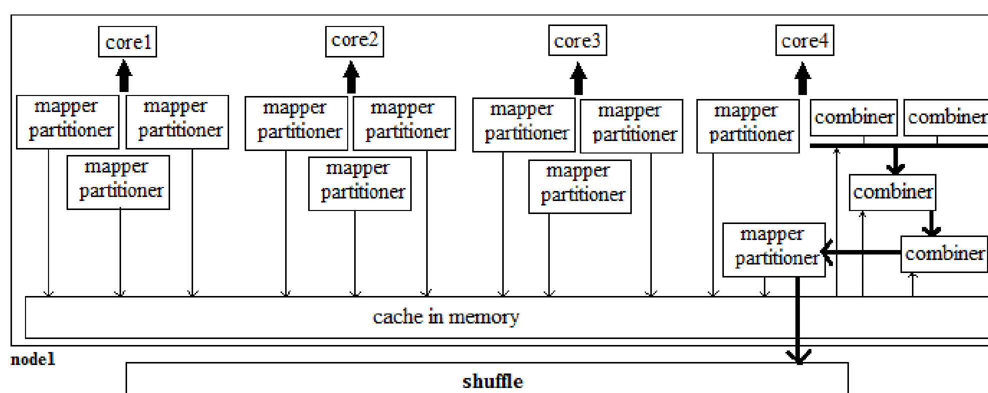


Figure 3.3 MLPNC

Algorithm 5: Minimize the number of intermediate records for wordcount job

```
1  Input parameters initialization:
2  Mem − Memcache
3  static HashMap Map_timer=new HashMap()
4  S − total cache size in MB
5  T − threshold size in MB
6  P − current_cache_pointer
7  COMB_i − combiners
8  current_cache_pointer P=0
9  threshold_seconds=30
10 combiner_enabled=true
11 node_id − VM id
12 map_id − map task ID
13 current_time − time at which map task launched
14 MLPNC
15 Start this service in the node where map tasks are launched
16 while true do
17 |  if P>=T || size(T+1 to (P mod S))<=T then
18 |  |  launch x=COMB_i++, y=COMB_i++
19 |  |  x takes 0 to T/2 && y takes T/2+1 to T
20 |  |  write records from (P+1 mod S) to T
21 |  if P mod S == T/2 then
22 |  |  launch (COMB_i) taking ( P mod S)+1 to T/2
23 |  |  write records from (P+1 mod S) to T
24 |  if (current_time − current_time(node_id+map_id)) > threshold_seconds)
   |    || last_mapper then
25 |  |  emit(records) − > reducers
26 end
27 Map task
28 class Mapper<LongWritable, Text, IntWritable, Text> {
29 Function setup():
30 |  Map_timer.put((node_id+map_id), current_time);
31 |  connect to Mem in node_id from map_id
32 Function map(LongWritable key, Text value):
33 |  String val[]=tokenize value with space separation;
34 |  for 1 to end(val) do
35 |  |  Mem < − write(word,1);
36 |  end
37 Function cleanmap():
38 |  disconnect from Mem
39 }
40 Combiner function (COMB)
41 class Reducer<IntWritable, Text, IntWritable, Text > {      // combiner method
42 Function reduce(Text key, Iterable<IntWritable> values):
43 |  for k in key do
44 |  |  result=sum(all values in the key)
45 |  |  Mem < − write(key,result)
46 |  end
```

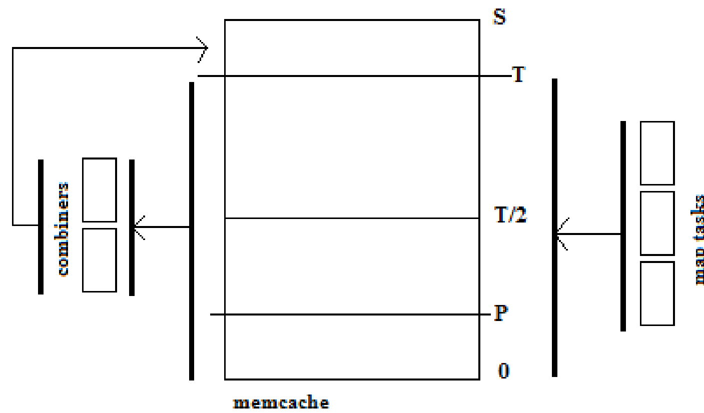


Figure 3.4 Storing intermediate records in Memcache

reaching threshold T or threshold seconds or until the last map task of a job arrives in the particular node. This whole process is controlled by MRAppMaster, which informs slave nodes about the last map task. Finally, the output records from cache are moved to shuffle phase by a current last map task. In the end of the map phase, the number of records is reduced as minimum as possible. Reduce tasks are also launched only when there is no further reduction is possible or last map task of a node arrives. So, resources are not held by reduce tasks for a long time.

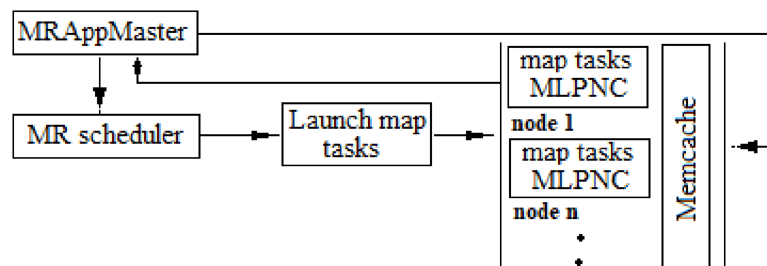


Figure 3.5 MLPNC system architecture

MLPNC component is run in every Hadoop VM as a separate service, as shown in Figure 3.5. Each node sends the map tasks status information to MRAppMaster for deciding which map tasks to send output records from the in-memory cache. Other advantages using an in-memory cache are:

- Consider four map tasks are running in a node. Assume map_1 did not produce more output records, but yet holding memory empty and map_4 generates more

records exceeding the reserved memory. Hence, memory usage varies across map tasks. However, map task latency is the same for all map tasks as they process every record once. Therefore, some map tasks hold memory without utilizing until the map task completion. By creating a common cache, such problems are overcome.

- As combiner is decoupled from map phase, map task latency is very less. It leads to achieve more number of data-local map tasks possible in a VM.

3.2 Results and Analysis

3.2.1 Dynamic Ranking based MapReduce Job Scheduler (DRMJS)

3.2.1.1 Experimental Setup

We evaluate our ideas on a testbed of eight PMs with different configuration and capacity to host VMs on KVM hypervisor. Table 3.2 lists out the configurations and the number of VMs hosted in each PM. 29 VMs are dedicated for Hadoop virtual cluster, and 14 VMs are for non-Hadoop VMs to trigger resource contention for Hadoop VMs as given in Table 3.3. We introduce random read/write (2 to 15 MB/s) to generate disk contention and random read/write (1 to 100 KB/s) to generate network contention via non-Hadoop VMs. Each PM's CPU performance is differentiated with different processor clock rate. Every VM is assigned with 2 virtual cores, 4 GB memory, 100 GB storage, and runs Ubuntu 16 OS. We also assume that each container size of MapRe-

Table 3.2 Physical Machines (PM) configuration

Class of PMs	Configuration of PM	VMs	VM configurations
PM_1	Intel(R) Xeon(R) CPU E5-2420 0, 1.90 GHz 6 cores, 32 GB memory, 1 TB HDD	8	2 virtual cores, 4 GB memory, 100 GB HDD
PM_2	Intel(R) Xeon(R) CPU E5-2680 v4, 2.40 GHz 28 cores, 132 GB memory, 3 TB HDD	23	2 virtual cores, 4 GB memory,100 GB HDD
PM_{3-4}	Intel(R) Core(TM) i5 CPU 650, 3.20 GHz 4 cores, 8 GB memory, 1 TB HDD	2x2	2 virtual cores, 4 GB memory, 100 GB HDD
PM_{5-8}	Intel(R) Core(TM) i7-3770 CPU, 3.40 GHz 4 cores, 8 GB memory, 1 TB HDD	4x2	2 virtual cores, 4 GB memory, 100 GB HDD

Table 3.3 Hadoop virtual cluster

PM	number of VMs	number of Hadoop VMs	number of non Hadoop VMs
PM_1	8	5	3
PM_2	23	16	7
PM_3	2	1	1
PM_4	2	2	0
PM_5	2	1	1
PM_6	2	1	1
PM_7	2	2	0
PM_8	2	1	1
Total	43	29	14

duce job holds 1 virtual core, and 1 GB memory. Local network bandwidth of VMs is 1 Gbps, and storage data rate is over 40 MB/s. We used workloads (wordcount, sort, wordmean) to experiment on PUMA [7] Wikipedia (150 GB) dataset. Despite dataset size is small to go for Hadoop, our intention is to demonstrate the ideas we have. HDFS block size is 128 MB, which results in 1200 blocks for 150 GB dataset. Therefore, there can be 1200 map tasks and we used 200 reduce tasks. We modify Fair scheduler to incorporate DRMJS and compared with default Fair scheduler [22] (resource-aware), and interference-aware scheduler [34].

3.2.1.2 Map and reduce task scheduling based on performance rank

In this section, we compared and contrasted four different cases on a set of workloads (wordcount, sort, wordmean) concerning map task latency, reduce task latency, and overall job latency on a heterogeneous environment. Case 1. Fair scheduler with no co-located VM's interference [22], Case 2. Fair scheduler with co-located VM's interference, Case 3. Interference-aware fair scheduler [34], and Case 4. DRMJS. Table 3.4 accounts the metrics we collected from these cases. It is observed that average map and reduce task latency of different workloads in case 2 increased over 200% and 75% respectively. Job latency peaked over 58% and makespan also doubled due to co-located VM's interference. It indicates that default resource-aware MapReduce scheduler performance is worse in multi-tenant environment. An interference-aware scheduler (Case 3) [34] considers historical interference pattern to predict the interference degree and

Table 3.4 Average map/reduce task latency for different workloads on heterogeneous environment

<table>
<tr><th>Job</th><th>Average map latency</th><th>Average reduce latency</th><th>Job latency</th><th>Makespan</th></tr>
<tr><td colspan="5">Case 1: Fair scheduler with no interference [22]</td></tr>
<tr><td>wordcount</td><td>8.9</td><td>73.5</td><td>420</td><td rowspan="3">1289</td></tr>
<tr><td>sort</td><td>12.3</td><td>91.3</td><td>681</td></tr>
<tr><td>wordmean</td><td>9.1</td><td>67.2</td><td>403</td></tr>
<tr><td colspan="5">Case 2: Fair scheduler with co-located VM's interference</td></tr>
<tr><td>wordcount</td><td>21.9</td><td>141</td><td>787</td><td rowspan="3">2479</td></tr>
<tr><td>sort</td><td>29.2</td><td>149.5</td><td>991</td></tr>
<tr><td>wordmean</td><td>27.4</td><td>121.1</td><td>709</td></tr>
<tr><td colspan="5">Case 3: Interference-aware scheduler [34]</td></tr>
<tr><td>wordcount</td><td>16.5</td><td>117.7</td><td>700</td><td rowspan="3">2189</td></tr>
<tr><td>sort</td><td>28.9</td><td>119.3</td><td>919</td></tr>
<tr><td>wordmean</td><td>23.4</td><td>109.7</td><td>668</td></tr>
<tr><td colspan="5">Case 4: DRMJS</td></tr>
<tr><td>wordcount</td><td>11.7</td><td>73.9</td><td>463</td><td rowspan="3">1479</td></tr>
<tr><td>sort</td><td>21.7</td><td>94.1</td><td>713</td></tr>
<tr><td>wordmean</td><td>15.3</td><td>64.2</td><td>419</td></tr>
</table>

avoids scheduling tasks in such nodes. It achieved 14% and 16% improvements in map and reduce task latency respectively in average of different workloads. But, there is no significant reduction in average job latency (9%) and makespan (12%). It is because, they consider past resource usage pattern of physical and virtual machines. It is not useful for increasing size of workloads and requires some extra work to predict co-located non-Hadoop VM's resource usage behaviour. To precisely predict co-located VM's resource usage behavior, we need to know the type of applications running. It is not possible due to tight isolation and privacy of VMs. Moreover, the dynamic performance of Hadoop VMs also is not considered. Therefore, we designed DRMJS that dynamically (every 30 seconds) calculates the performance of Hadoop VMs. The performance score is calculated for map and reduce task separately for each VM, unlike previous methods discussed in the literature survey that calculate node performance for both map and reduce tasks in common. It is essential to calculate the performance of map/reduce tasks in every VM separately as they demand different resources during their execution. As shown in Figure 3.6, Figure 3.7 and Figure 3.8, DRMJS improved

average map task latency of wordcount, sort, and wordmean jobs up to 30%, 25%, and 26% respectively than Case 3. Similarly, average reduce task latency of wordcount, sort, and wordmean jobs improved up to 38%, 22%, and 42% respectively than Case 3. DRMJS improved overall job latency up to 34%, 23%, 38% for wordcount, sort, and wordmean jobs, respectively than Case 3, as shown in Figure 3.9. Ultimately, makespan also improved over 33% than Case 3, as shown in Figure 3.10.

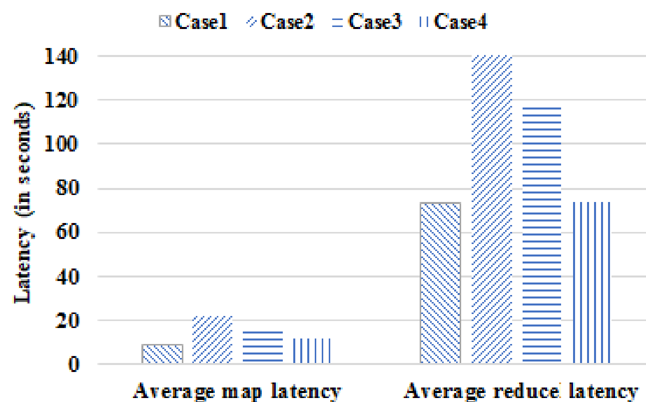


Figure 3.6 Average map/reduce task latency of wordcount job with different cases

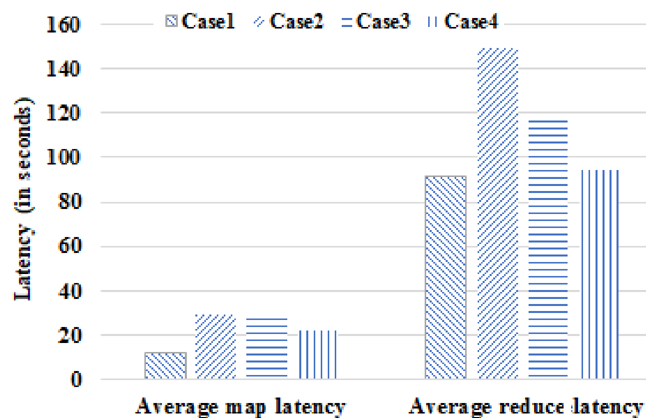


Figure 3.7 Average map/reduce task latency of sort job with different cases

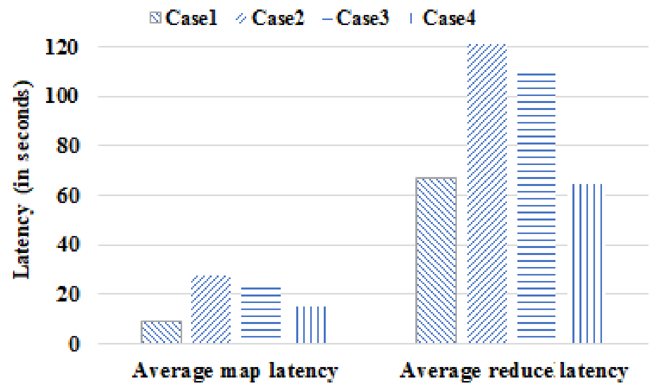


Figure 3.8 Average map/reduce task latency of wordmean job with different cases

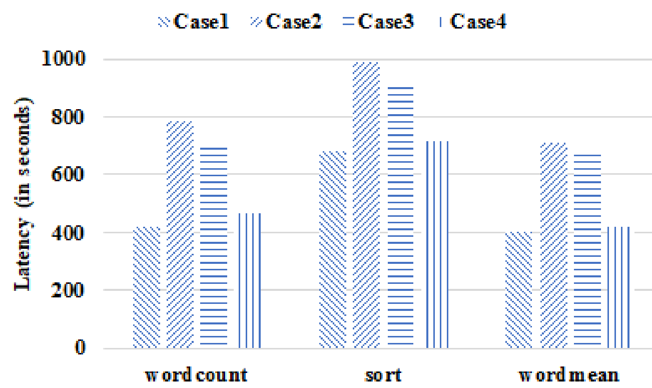


Figure 3.9 Job latency of different workloads with different cases

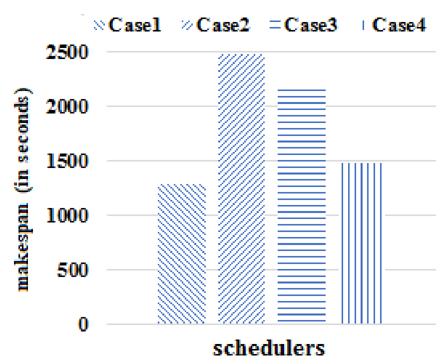


Figure 3.10 Makespan of different cases

3.2.1.3 Scheduling reduce task based on its input size and performance rank

In the previous section, we discussed the allocation of the map and reduce tasks by understanding the dynamic performance of VMs. Input size of map tasks is always the

Table 3.5 Number of reduce tasks and its average latency for wordcount job using Case 3 and Case 4

S. No.	reduce input size (GB)	number of reduce tasks	Case 3 average reduce latency (in seconds)	Case 4 average reduce latency (in seconds)
1	1.9	16	242	173
2	1.7	15	212	151
3	1.6	13	221	148
4	1.5	11	190	134
5	1.3	7	231	120
6	1.1	9	167	90
7	1	10	148	91
8	0.9	7	130	87
9	0.8	13	105	60
10	0.7	12	95	52
11	0.6	15	57	38
12	0.5	13	68	41
13	0.4	9	42	20
14	0.3	18	31	17
15	0.2	15	29	15
16	0.1	17	23	14

same. However, the input size of reduce tasks is heterogeneous. At times, reduce task taking very less input may be allocated to high performing VM (C_1) or reduce task taking huge input may be allocated to low performing VM (C_4). The later one may lead to reduce task to be a straggler, which ultimately prolongs overall job latency. Case 4 handles this problem in an elegant way. After applying Algorithm 1, MRAppMaster classifies (Algorithm 4) Hadoop VMs into four performance classes (C_1,C_2,C_3,C_4) based on the reduce performance of each VM. We then dynamically find the total size of reduce task input (adding up all partitions available in all map tasks) and map them with any one of the four classes (C_1,C_2,C_3,C_4) of reduce node performance. Classifying nodes like this is important, because, relatively equal performance nodes are put into the same class. So, even if there is no container possible in a node of C_1, another node in the same class can be inspected for containers. If none of the nodes in the same class have resources, the successive classes are inspected to launch containers.

We discuss reduce task allocation with Case 3 and Case 4 based on three parameters

(Table 3.5): reduce task's input size, number of reduce tasks, and an average latency of reduce task having same input size. From Table 3.5, we can observe that there are different reduce tasks taking different size of the input. For instance, reduce input size varies from 1.9 GB to 0.1 GB. Case 3 launched 15 reduce tasks each taking 1.7 GB input with an average job latency of 212 seconds. To understand average job latency variation, consider Case 3 that launches 7 reduce tasks taking 1.3 GB input. It takes 231 seconds on average to complete, which is almost equivalent to 13 reduce tasks taking 1.6 GB input. It is because, Case 3 allocates reduce tasks just by observing the interference degree (level of interference) of VMs, but not the amount of work each reduce task does. Therefore, it is essential to consider the amount of reduce tasks input along with the dynamic performance of Hadoop VM to minimize latency further. Case 4 (DRMJS) minimized average task latency over 28%-50% than Case 3 (Figure 3.11). It is because, Case 4 rightly placed reduce tasks considering its input size and allocates onto the right VM by considering dynamic performance.

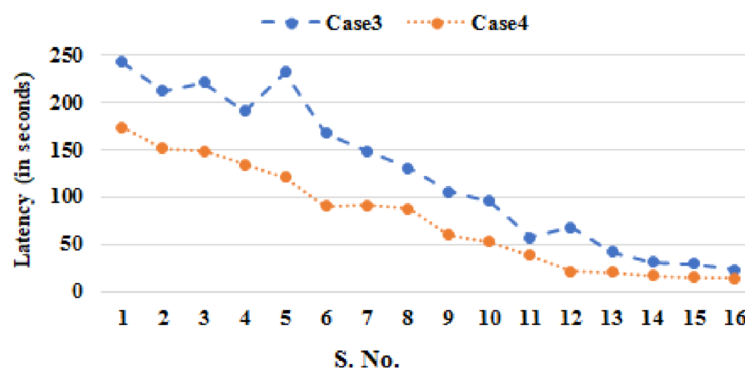


Figure 3.11 Average reduce task latency of Case 3 and Case 4 for wordcount job

3.2.1.4 Performance score vs the number of map/reduce tasks in a Hadoop VM

While scheduling map/reduce tasks based on the rank of VMs, we tracked how many number of map and reduce tasks was placed according to the performance score. For instance, when a VM has high reduce performance score, then reduce task is preferred to launch as map latency will be prolonged due to interference. After finding VMs rank, it is mapped with one of the four classes (C_1,C_2,C_3,C_4) to choose the right VM for re-

duce tasks. Figure 3.12 exhibits the various cases of map and reduce performance score of VMs. Let us consider a VM taking 12 different possible performance score com-

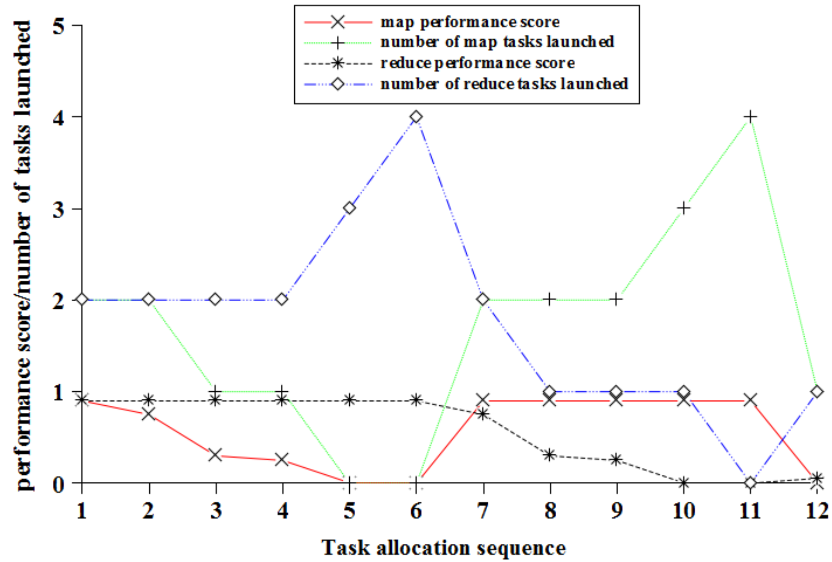


Figure 3.12 Performance score vs number of map/reduce tasks allocated

binations in map/reduce task allocation sequence. Consider task allocation sequence 1.When map and reduce performance score are high (0.9) then particular VM has no interference, and a maximum number of containers for map/reduce tasks can be allocated. When the performance score of map and reduce task is less than 0.008, then there is no scope to launch a map or reduce task. When map performance score is high (0.9) and reduce performance score is low (0.007) as in task allocation sequence 11, map tasks are assigned subsequently satisfying data locality. Table 3.6 lists out the number of map and reduce tasks avoided from being allocated to a node where co-located VM's interference is possible. For instance, PM_1 avoids over 90% map tasks, and 100% reduce tasks, while PM_2 avoids 92% map tasks, and 84% reduce tasks. A map/reduce task latency prolongs when it takes more time than the average map/reduce latency of the current job. The interesting point is that the percentage of map/reduce tasks being avoided from interference increases while number of non-Hadoop VMs increase in a physical host. Because the possibility of allocating container for tasks without interference goes less likely when a number of co-located VM increases. PM_4 and PM_7

Table 3.6 Avoidance of map/reduce tasks from interference

PM wise task allocation	Case 3		Case 4	
	number of map tasks prolonged its latency	number of reduce prolonged its latency	number of map tasks avoided the interference	number of reduce tasks avoided the interference
PM_1	23	11	21	11
PM_2	51	19	47	16
PM_3	4	2	4	2
PM_4	9	3	8	1
PM_5	3	2	3	1
PM_6	5	2	5	1
PM_7	8	4	8	3
PM_8	5	2	4	2

do not have co-located VMs. Therefore, it is decided to schedule tasks based on the performance of VM.

3.2.1.5 Resource utilization

As shown in Figure 1.12, utilization of other resources is minimized gradually when IO resource enters into bottleneck. Figure 3.13 shows the improved resource utilization wne disk IO is prone to bottleneck. With DRMJS, CPU utilization and N/W IO improved 30%-65%, and 35%-60%, respectively, on average by understanding interfer-

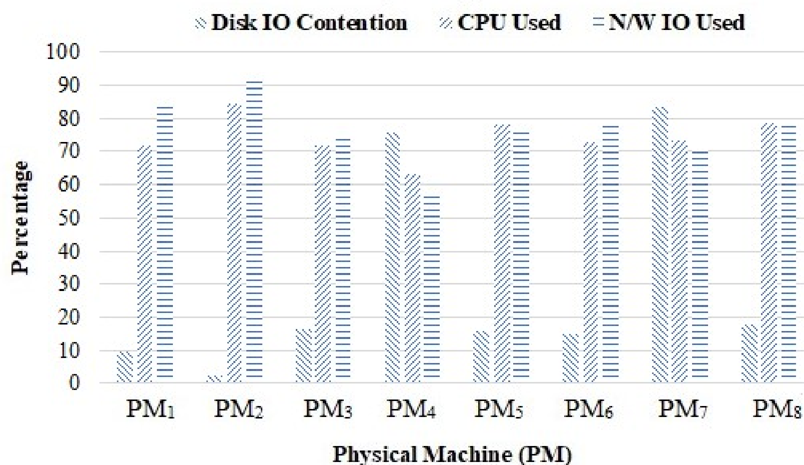


Figure 3.13 Resource utilization after DRMJS

ence of co-located VMs when disk IO contention exists. It has become possible as we allocate map and reduce tasks not only based on the hardware heterogeneity and also considering the co-located VM's interference.

3.2.2 Multi-Level Per Node Combiner (MLPNC)

3.2.2.1 Experimental Setup

We experimented and evaluated MLPNC on a testbed with the following configuration: physical server with Xeon processor, KVM hypervisor, VMs with the configuration of dual-core (hyper-threaded), 8 GB memory, 50 GB HDD, and 1 Gbps full duplex network link, Linux OS, and Hadoop 2.7.0. We preferred wordcount job for the experiment, which is the best candidate to evaluate the shuffle phase as the number of records emitted by map tasks are greater than its input. We tested a wordcount job on three different virtual clusters with 2, 3, and 4 nodes (VMs). Each VM is hosted in different physical server (in the same cluster) to track the number of records prepared for the shuffle phase. We used three different sizes of text dataset: 10 GB, 15 GB, 20 GB with default HDFS block size 128 MB. These text datasets are generated by the pre-defined MapReduce job "randomtextwriter" that comes along with Hadoop distribution. A number of map tasks launched for 10 GB, 15 GB, 20 GB input dataset are 90, 135, and 180 respectively. Number of reduce tasks are 5 for all experiments. Since each VM contains 8 GB memory, there can be 7 containers (each with 1 virtual core and 1 GB memory) for map tasks possible at any point of time. For reduce tasks, we allocate a container with 1 virtual core and 2 GB memory. We considered two methods to compare the efficiency of MLPNC: default combiner, PNC [29]. Comparisons among these approaches are extensively presented based on number of shuffled records, average shuffle latency, average merge latency, and average reduce task start latency.

3.2.2.2 Results and Analysis

MLPNC minimizes the number of shuffle records by running combiner function multiple levels until there is no further reduction possible or last map task arrives in a node. Figure 3.14 illustrates the number of shuffled records (in millions) produced by different approaches for different size of datasets on two nodes (Figure 3.14(a)), three nodes

(Figure 3.14(b)), four nodes (Figure 3.14(c)). MLPNC produced less number of records than other approaches such as default combiner and PNC. MLPNC minimized the number of shuffled records up to 33% compared to PNC. Shffule phase is responsible for the major portion of job latency. As there is a significant drop in the number of shuffled records, shuffle phase latency also minimized significantly, as shown in Figure 3.15, using MLPNC up to 40% compared to PNC. When the number of records received at reducer side is high, records have to be spilled into the local disk from memory. Once all records received from all map tasks, spilled files have to be merged in order to perform the sort. This merging time is dependent on the number of shuffled records. So, MLPNC minimized the shuffle latency up to 30% compared to PNC on average for different cluster size. This results in minimizing the reduce task latency up to 20-25% on average compared to PNC.

It is very important to launch reduce tasks at the right time to receive map output records. Launching reduce tasks early (along with map tasks) clogs up the cluster re-

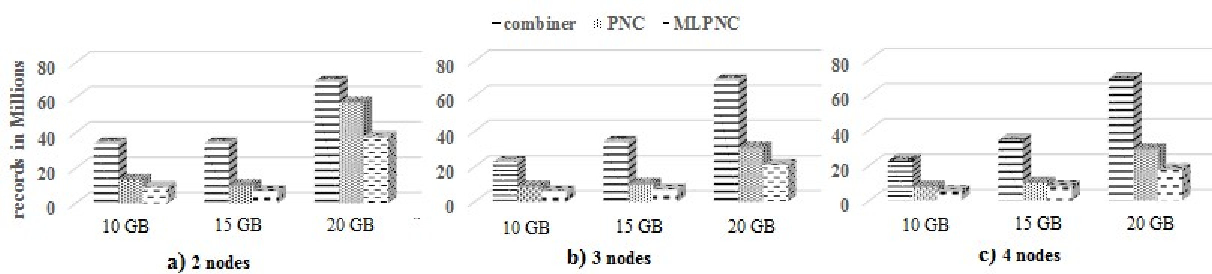


Figure 3.14 Number of shuffled records generated by different approaches for different sizes of dataset

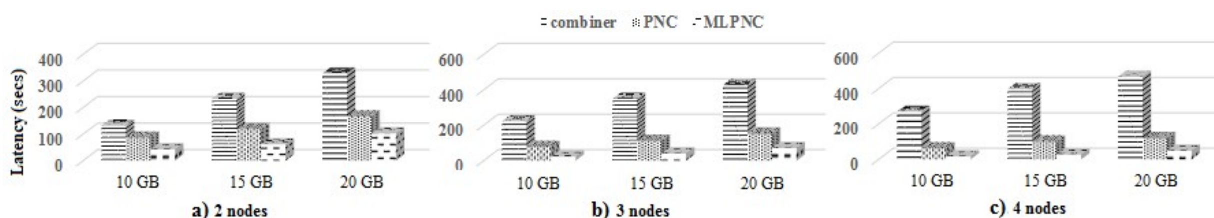


Figure 3.15 Average shuffle latency using different approaches for different sizes of dataset

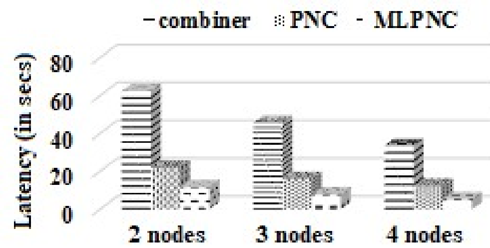


Figure 3.16 Reduce task start latency for all datasets based on different approaches

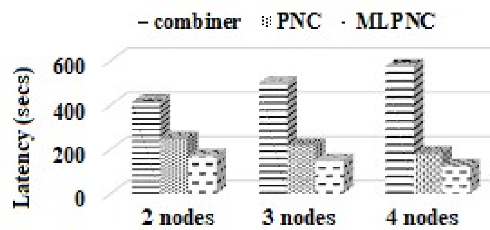


Figure 3.17 Overall job latency

sources for a long time. Launching reduce tasks after all the map tasks completed leads to massive network bandwidth consumption, which increased job latency. Because all map nodes start transferring the intermediate results to reduce nodes at the same time. MapReduce has properties to customize when to launch reduce tasks. However, in PNC, when cache hits threshold or last map tasks launched in a VM, intermediate records are moved to reduce nodes. So, it requires reduce tasks to be launched once any of the nodes running PNC reach cache threshold. In contrast, MLPNC minimizes the number of intermediate records level by level until there is no further reduction is possible or last map task of that node arrived. Figure 3.16 shows that MLPNC launches reduce tasks after 10-15 seconds on average than PNC. Consequently, MLPNC minimized overall job latency up to 32% on average, as shown in Figure 3.17, compared to PNC for different datasets on a different number of nodes in the cluster.

3.2.3 Reduce task scheduling based on performance rank after MLPNC

3.2.3.1 Experimental Setup

We evaluated this idea with Hadoop 2.7.0 on a testbed with eight different physical machines of different configuration and capacity to host VMs using KVM hypervisor. Table 3.2 lists the PM's configuration and number of VMs hosted in each PM. As given

in Table 3.3, 29 VMs are dedicated for Hadoop virtual cluster, and 14 VMs are for non-Hadoop VMs to trigger resource contention for Hadoop VMs. We introduced random read/write (5 to 15 MB/s) to generate disk contention and random read/write (1 to 100 KB/s) to generate network contention via non-Hadoop VMs. Each PM's CPU performance is differentiated with different processor clock rate. Every VM is assigned with 2 virtual cores, 4 GB memory, 100 GB storage, and runs Ubuntu 16 OS. We also assume that each container size of MapReduce job holds 1 virtual core and 1 GB memory. Local network bandwidth of VMs is 1 Gbps, and storage data rate is over 15 MB/s. We used PUMA [7] Wikipedia (150 GB) dataset to experiment with wordcount job further. Despite dataset size is small to go for Hadoop, our intention is to demonstrate the idea we proposed. HDFS block size is 128 MB, which results in 1200 blocks for 150 GB dataset. Therefore, there can be 1200 map tasks (1 map task for each block) for the wordcount job.

3.2.3.2 Results and Analysis

Even though DRMJS minimized the latency of map and reduce tasks, yet there is a possibility to minimize reduce task latency by understanding the size of each reduce task's input. While scheduling reduce tasks, nodes for reduce tasks are selected from the reduce rank list based on the size of reduce task's input. We demonstrated reduce task allocation and its latency based on 4 cases: Case 1. job with no combiner, Case 2. job with a combiner, Case 3. job with MLPNC, and Case 4. job with MLPNC using DRMJS. We compared and contrasted each case with 3 parameters: number of reduce tasks, reduce task's input size, and average latency of reduce tasks having the same input size. For all these cases, we set a constraint that the number of reduce tasks depends on the size of overall map tasks output. Table 3.7 accounts the number of reduce tasks launched for first three cases. For example, if map phase output is 4 GB, then 8 reduce tasks (assuming 500 MB input for each reduce task) are decided, to avoid a single reduce task dumped with huge input. At times, before some map tasks get over, we have to launch reduce tasks. In this case, the number of reduce tasks is determined based on the overall map phase output dynamically. This constraint is to decide the number of reduce tasks, and nothing to do with reduce tasks input size. This constraint

Table 3.7 Number of reduce tasks for all cases

Different cases	Total map phase output (in GB)	Number of reduce tasks
job input	150	
Case 1	178	356
Case 2	81.2	163
Case 3	53.2	107

is required because, when we launch 200 reduce tasks for Case 1, each reduce task will have a large size of the input. If we run 200 reduce tasks for Case 3, then most of the reduce tasks will do very little work, which leads to resource under-utilization.

Table 3.8 records reduce input size, number of reduce tasks with the same input size, and its average latency for all four cases. From Table 3.8, we can observe that there are different reduce tasks taking the different size of the input. For instance, for Case 1, reduce input size varies from 1.9 GB to 0.1 GB. Seven reduce tasks take 1.9 GB input with average latency of 191 seconds. Average reduce task latency does not decrease linearly with decreasing reduce tasks input size (Figure 3.18). It is because most of the reduce tasks taking less input size are sprawled across a virtual cluster. This point is valid for Case 2, as shown in Figure 3.19. Similarly, for Case 1, the average latency of 10 reduce tasks taking 1.2 GB input is 167 seconds.

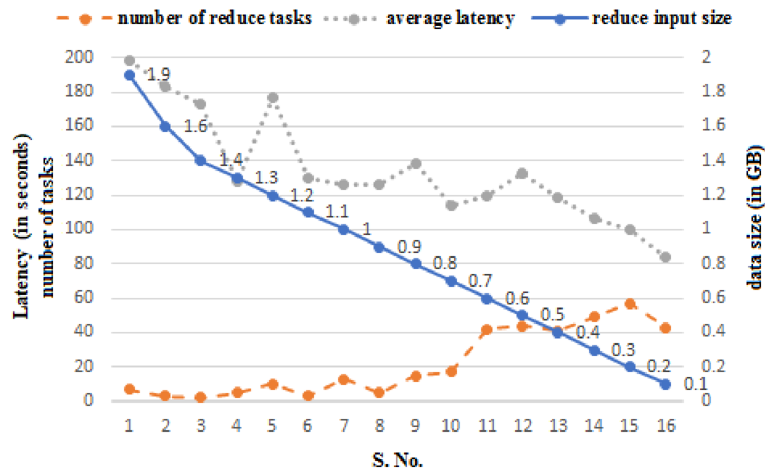


Figure 3.18 Case 1: Reduce task latency with no combiner

Table 3.8 Number of reduce tasks and its average latency

S. No.	with no combiner (Case 1)			with combiner (Case 2)			with MLPNC (Case 3)			MLPNC with dynamic performance (Case 4)		
	reduce input size (GB)	number of reduce tasks	average latency (in seconds)	reduce input size (GB)	number of reduce tasks	average latency (in seconds)	reduce input size (GB)	number of reduce tasks	average latency (in seconds)	reduce input size (GB)	number of reduce tasks	average latency (in seconds)
1	1.9	7	191	1.5	8	179	1.7	2	189	1.7	2	137
2	1.6	3	180	1.4	2	187	1.5	3	173	1.5	3	121
3	1.4	2	171	1.2	11	171	1.4	4	183	1.4	4	119
4	1.3	5	123	1.1	7	133	1.1	7	156	1.1	7	107
5	1.2	10	167	0.9	9	169	1	5	167	1	5	99
6	1.1	3	127	0.7	7	101	0.8	7	127	0.8	7	93
7	1	13	113	0.6	12	89	0.6	11	113	0.6	11	73
8	0.9	5	121	0.5	5	83	0.4	3	87	0.4	3	61
9	0.8	15	123	0.4	21	71	0.3	25	71	0.3	25	53
10	0.7	17	97	0.3	19	53	0.2	21	51	0.2	21	49
11	0.6	42	78	0.2	25	37	0.1	19	39	0.1	19	37
12	0.5	44	89	0.1	37	31						
13	0.4	41	78									
14	0.3	49	57									
15	0.2	57	43									
16	0.1	43	41									

It is comparatively not much better than 2 reduce tasks taking 1.4 GB input that complete in 171 seconds. It is because reduce tasks are allocated arbitrarily regardless of the dynamic performance of VMs. The same scenario can be observed in Case 2 and Case 3 also, as shown in Figure 3.19. Therefore, it is important to understand the dynamic performance of each VM. MLPNC considers the input of reduce tasks from every VM rather than every map tasks. With MLPNC, at times, reduce task receiving 1.5 GB input takes more or equal time as reduce tasks processing 1 GB. It is because some of the reduce tasks processing 1.5 GB is allocated to low performing VM. Therefore, despite minimizing the number of intermediate records in the shuffle phase using MLPNC, there is a scope to improve job latency further by exploiting performance heterogeneity in a virtual environment. While exploiting dynamic performance for MLPNC, we initially find the total size of reduce phase input (adding up all partitions available in all VMs) and classifies them into any one of the four class of node performance (C_1,C_2, C_3, C_4). It is important because relatively equal performance nodes are put into the same class. So, even if there is no container possible in a node of C_1, another node that belongs to the same class can be inspected for containers. If none of the nodes in the same class has a container, then the successive class is inspected for containers. Assigning reduce tasks that belong to C_1 to other lower classes may result in a straggler. However, reduce tasks that belong C_4, can be assigned with any of the upper classes.

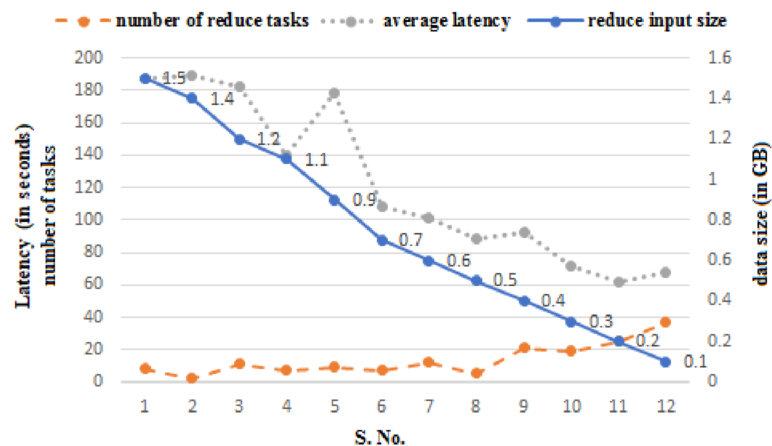


Figure 3.19 Case 2: Reduce task latency with combiner

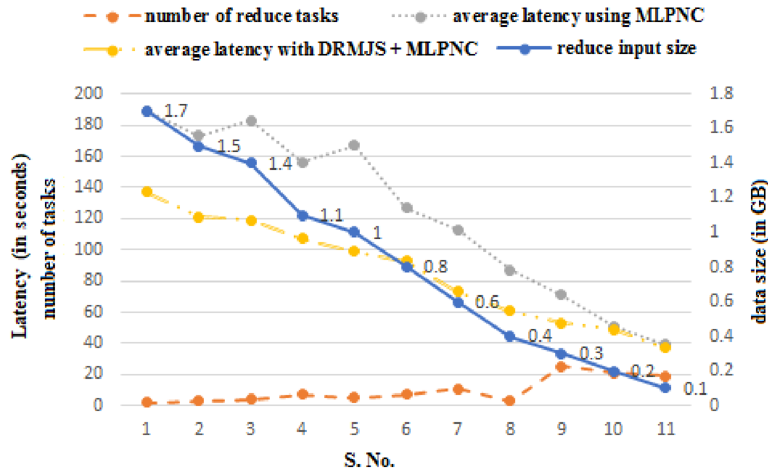


Figure 3.20 Reduce task latency with dynamic performance vs MLPNC

Figure 3.20 shows the comparison between MLPNC and MLPNC with dynamic performance. It is observed that the latency of particular reduce tasks having bigger input size is vastly reduced. For instance, reduce task taking 1.7 GB input takes 28% less time than MLPNC with no dynamic performance. Similarly, 28%-41% improvement in reduce task latency is achieved for reduce tasks taking input from 1.7 GB to 0.6 GB as they will have more probability to get allocated with high performing nodes class (C_1, C_2). However, there is no much improvement in average reduce latency with an input size of fewer than 0.5 GB. It is because the number of reduce tasks taking less than 0.5 GB input size is high, and they are sprawled in all nodes in the virtual cluster.

3.3 Summary

The primary concerns for Hadoop users are improving job latency and resource utilization. However, Hadoop virtual cluster faces many challenges as they are spread across racks in CDC and co-located with non-Hadoop VMs. One of the significant problems is the heterogeneous performance of VMs due to hardware heterogeneity and co-located VM's interference. It largely impacts job latency and resource utilization. So, we proposed DRMJS to improve job latency and resource utilization in a virtual cluster. DRMJS dynamically observe the performance of a VM for map and reduce tasks sepately. Ultimately, DRMJS improved overall job latency, makespan, and resource utilization

up to 30%, 28%, and 60%, respectively, on average compared to existing MapReduce schedulers. To minimize job latency further, we minimized the number of intermediate records using MLPNC. MLPNC outperformed PNC up to 33% reduction in a number of shuffled records, and up to 32% reduction in average job latency. We also placed reduce tasks based on its input size by exploiting heterogeneous performance after MLPNC. Results claimed that 28%-41% of reduce task latency improvement is possible by exploiting performance heterogeneity.

Chapter 4

MapReduce Job Scheduling

4.1 Proposed Methodologies

From the outcome of literature survey, we can understand that yet there are opportunities for Hadoop MapReduce to improve makespan and resource utilization by scheduling heterogeneous jobs in heterogeneous virtualized environment. The proposed works are:

1. Roulette Wheel Scheme (RWS) based data block placement in HDFS to minimize job latency.

2. Constrained 2-dimensional bin packing map/reduce tasks using Ant Colony Optimization (ACO).

3. Fine-Grained Data-Locality Aware (FGDLA) job scheduling to minimize the number of intermediate records for a batch of jobs.

4.1.1 Roulette Wheel Scheme (RWS) based data block placement

We developed a model that helps to place data blocks across Hadoop VMs based on the capacity of VM flavors. In general, data is divided into equal sized (128 MB) chunks, called blocks, before loading onto HDFS. By default, the replication factor of data blocks in HDFS is 3. So, three copies of a data block are stored based on the topology awareness in the physical cluster. However, topology awareness in virtual cluster is not yet known to be implemented in HDFS. Therefore, every VM will receive the same number of data blocks at any point of time. Consider v number of VMs (VM_1,

VM_2,..., VM_v) in the Hadoop virtual cluster. If the data size is t TB and default data block size is 128 MB, then the number of data blocks n is calculated as $n = t/128MB$. If the replication factor is 3, the total number of data blocks is $n*3$. These blocks are placed across VMs equally, which means that each VM will get $(n*3)/v$ blocks. Now, consider five different VM flavors (VMF_1, VMF_2, VMF_3, VMF_4, VMF_5), as given in Table 4.1, with different resource capacity. In VMF_1, at any point in time, there can be only one block processed or one map/reduce task can be launched. In VMF_5, up to 12 different tasks can be launched if each container requires exactly 1 vCPU. For instance, at time C_t, VMF_5 will be able to complete over 10 map/reduce tasks while VMF_1 can finish only one map/reduce task. If every VM has to process an equal number of blocks, VM flavors containing more capacity will complete more number of blocks in parallel compared to VMF_1. At this moment, VMF_5 becomes idle as there are no more blocks to process but VMF_1 has to finish processing some more blocks. Now, blocks from VMF_1 are copied into VMF_5 for non-local execution. This incurs network bandwidth to transfer those unprocessed data blocks over the network. Therefore, the latency of a job increases, which in turn increases the makespan.

In order to minimize the makespan, we have to place blocks based on the capacity of individual VM. This means that the number of data blocks loaded into each VM depends upon the number of vCPU allocated to VMs considering a task demands one core at any time. Placing blocks with this idea, when VMF_1 completes one task, VMF_5 can complete as maximum as possible, thus parallelism is achieved. We used a probability method, Roulette Wheel Scheme (RWS), to find the number of blocks that can be placed in each VM flavor based on its resource capacity. RWS is a French word meaning "little

Table 4.1 Percentage of blocks to store in different VM Flavours

VMF_f	Flavor name	$VMF_f < vCPU, Memory, Storage >$	P_f
VMF_1	small	$VMF_1 < 1,2,20 >$	1/27=0.037%
VMF_2	medium	$VMF_2 < 2,4,40 >$	2/27=0.07%
VMF_3	large	$VMF_3 < 4,8,80 >$	4/27=14%
VMF_4	x large	$VMF_4 < 8,16,160 >$	8/27=29%
VMF_5	2x large	$VMF_5 < 12,24,250 >$	12/27=44%

wheel". The concept is, a wheel is divided with pie-cut pizza shape. If we rotate this wheel with a small ball rolling on it, the probability of ball resting on the large pizza-cut is high when the wheel stops rotating. Similarly, to place data blocks, we consider the number of vCPUs in a VM flavor. Therefore, VM having more vCPU will attract more number of blocks. Equation 4.1 generalizes this statement based on the number VM flavors given in Table 4.1.

$$\forall_f, P_f = \frac{VMF_f < vCPU >}{\sum_{z=1}^{5} VMF_z < vCPU >} \tag{4.1}$$

We consider only the available number of vCPU in each VM flavor as map/reduce tasks are space shared (not time shared) and constitute the number of containers that can be formed in a VM. Equation (4.1) adds up all the vCPU allocated in each flavor and calculates the relative percentage of data blocks to store in each VM flavor. Therefore, each VM that belongs to VMF_1 gets 0.037% of overall blocks intended to be stored in HDFS. Similarly, VMF_2, VMF_3, VMF_4, VMF_5, get 0.07%, 14%, 29%, and 44% of overall data blocks stored in HDFS as given in Table 4.1. However, this model is applicable only when the replications of a data block are stored in the VMs of the same flavor. In general, different copies of a data block are placed in different VMs regardless of the VM flavor. For instance, consider a dataset with 1000 blocks ($b_i | i = 1 \quad to \quad 1000$). If the replication factor is 3, there will be 3000 blocks. As there is no rack awareness in the virtual environment, copies of a data block may be stored in two different cases:

- *Case 1*: If block b_1 is stored in different VMs of the same flavor, in which VM should map task for block b_1 be executed? It is done in any VM of the same flavor. This does not affect the overall probability of map task execution of a VM flavor.

- *Case 2*: If block b_1 is stored in VMs of different flavors (say, VMF_1, VMF_3, and VMF_5), in which VM flavor should map task for block b_1 be executed? Now, executing in any of the VM flavors affect the overall probability of map tasks processed in each VM flavor.

Therefore, if a block b_1 is stored in VMF_1, then its replica should be stored in the VM

of same flavor satisfying data locality. This is the major idea in our block placement strategy. Achieving this, every VM flavor has a P_f probability of blocks to be processed for each workload. Moreover, non-local execution is avoided as maximum as possible.

4.1.2 Constrained 2-dimensional bin packing map/reduce tasks

Bin packing tasks is largely studied in a heterogeneous virtual environment. However, bin packing concept is entirely new for map/reduce task scheduling to the best of our knowledge. Placing map/reduce tasks of different jobs affects resource utilization and makespan. Therefore, MapReduce scheduler should be aware of VM flavors and its capacity in order to place the right combination of map/reduce tasks of heterogeneous jobs. To achieve this, we transform the map/reduce task scheduling problem into 2-dimensional constrained bin packing problem.

4.1.2.1 Challenges

Two major challenges are considered to improve the makespan and resource utilization of a MapReduce job scheduler.

- Heterogeneous jobs (varying size container, varying number of tasks).
- Heterogeneous VM capacities.

4.1.2.2 Problem definition

"Placing map/reduce tasks of heterogeneous jobs onto bins (VMs) of heterogeneous capacities to improve makespan and resource utilization" is represented as Equation 4.2.

$$F : \forall_j, \forall_i, Comb_{J_j} < map, reduce > \rightarrow B_i^{h,g} \quad (4.2)$$

n map tasks and m reduce tasks of a job (J_j) may be executed at any point of time in a bin ($B_i^{h,g}$) obeying the constraints of map and reduce tasks. A combination of map and reduce tasks of a job executed in a bin is denoted as ordered pair $Comb_{J_j} < n*map, m*reduce >$. A bin may execute any combination of map/reduce tasks (for instance, $Comb_{J_1} < 5,0 > \wedge Comb_{J_2} < 0,2 > \wedge Comb_{J_3} < 0,0 > \wedge \ldots \wedge Comb_{J_M} < 3,0 >$) of M different jobs. However, it is not certain that map/reduce tasks of all the jobs should be executed in a bin at any time. Because, if there are no data blocks to be

processed by a job in a bin, then map/reduce tasks of that particular job will not be included in this pair sequence.

4.1.2.3 Notations used

We list some of the notations and its descriptions used in our problem formulation.

X - Number of racks in a CDC.

R_g - g^{th} rack $\{R_1, R_2 ... R_g ... R_X\}$.

Y - Number of PMs in each R_g.

P_h^g - h^{th} PM in g^{th} rack $\{P_1^g, P_2^g ... P_h^g ... P_Y^g\}$.

N - Number of VMs (bins).

$B_i^{h,g}$ - i^{th} bin in h^{th} PM in g^{th} rack $\{B_1^{h,g}, B_2^{h,g} ... B_i^{h,g} ... B_N^{h,g}\}$.

M - Number of workloads (jobs).

J_j - j^{th} job $\{J_1, J_2 ... J_j ... J_M\}$.

$BR_i^{v,u}$ - Bin Resource: Number of vCPU (v) and amount of memory (u) in i^{th} bin.

$Task_j^{p,q}$ - Number of map tasks (p) and reduce tasks (q) of each J_j

$AR_M_{i,j}^{v,u,r}$ - Allocated Resources (v vCPU, and u Memory) for r^{th} map task of J_j in i^{th} bin.

$AR_R_{i,j}^{v,u,s}$ - Allocated Resources (v vCPU, and u Memory) for s^{th} reduce task of J_j in i^{th} bin.

T_i - Number of map/reduce tasks allocated in i^{th} bin.

$Comb_{J_j} < map, reduce >$ - Combination of map/reduce tasks of different J_j in a bin.

C_t - Commit time: launching map/reduce tasks of different jobs every t seconds.

4.1.2.4 Objective function

Our main objective of bin packing map/reduce tasks is to improve the resource utilization for a batch of heterogeneous jobs in heterogeneous bin capacities. As a result of packing map/reduce tasks as much as possible utilizing all the resources of a virtual cluster, makespan also could be minimized as a by-product. However, individual job completion time could vary. For instance, the completion time of the shortest job could be higher than a bigger job in the batch. Because we submit a batch of jobs, where

individual job latency is ignored to improve resource utilization. We initially find different possible combinations $< map, reduce >$ for each job in every bin. For instance, consider two jobs (J_1, J_2), 100 VMs each with 4 containers. For simplicity, we consider only map tasks of jobs in finding combinations for each bin, because only after all map tasks completed, reduce tasks are launched. Therefore, possible combinations of map/reduce tasks of jobs (J_1, J_2) in all bins are $< 4,0 >< 0,0 >, < 3,0 >< 1,0 >$ $, < 2,0 >< 2,0 >, < 1,0 >< 3,0 >, and < 0,0 >< 4,0 >$. As we are going to modify fair scheduler to incorporate bin packing scheme, we need to ensure a fair share of resources among jobs. For instance, $< 4,0 >< 0,0 >$ from $B_1^{h,g}$, $< 2,0 >< 2,0 >$ from $B_2^{h,g}$, $< 1,0 >< 3,0 >$ from $B_3^{h,g}$, etc. After finding these combinations, we need to evaluate whether the resources in each bin are utilized the maximum or not using Equation 4.3, which comprises two components (vCPU, and memory) to find Total Allocated Resource in i^{th} bin (TAR_i).

Total Allocated Resource in $B_i^{h,g}$=TAR_i=

$$\frac{\sum_{k=1}^{T_i} AR_M_k^v \vee AR_R_k^v}{BR_i^v} \times \frac{\sum_{k=1}^{T_i} AR_M_k^u \vee AR_R_k^u}{BR_i^u}, \quad at \quad C_t \tag{4.3}$$

To find the utilization of vCPU of a bin, we find the ratio between the number of vCPU occupied by all tasks (T_i) running in the ith bin at present and the total number of vCPU available in the bin. Similarly, we calculated the utilization of memory as well. If any one of the resources is utilized very less, for example, 90% (vCPU) and 10% (memory), then TAR_i will be just 0.09 (0.9 x 0.1), which is not desired. Therefore, for all the combinations found in each bin, TAR_i is calculated as given in Table 4.2. We only consider the combinations that result in over 90%. After calculating TAR_i, our objective is to improve the resource utilization in individual bin, and overall resource utilization. Equation 4.4 calculates the Utilization of Individual Bin (UIB) for different combinations while the Overall Cluster Resource Utilization (OCRU) is calculated by Equation 4.5. Table 4.2 lists out the combinations of different jobs for VMs of different flavor.

$$UIB = Min(1 - TAR_i) \tag{4.4}$$

$$OCRU = Min \sum_{i=1}^{N} (1 - TAR_i) \tag{4.5}$$

Table 4.2 Possible combination of map tasks of different jobs in a VM

J_1	J_2	J_3	J_4	J_5	J_6	Resource utilization (TAR)
$VMs \in VMF_1$						
1	0	0	0	0	0	100
0	0	0	0	1	0	100
$VMs \in VMF_2$						
2	0	0	0	0	0	100
1	0	0	0	1	0	100
0	0	1	0	1	0	93.75
1	0	1	0	0	0	93.75
$VMs \in VMF_3$						
4	0	0	0	0	0	100
2	0	0	0	2	0	100
1	0	0	0	3	0	100
0	0	0	0	4	0	100
0	0	1	0	3	0	96.88
3	0	1	0	0	0	96.88
2	0	1	0	1	0	96.88
3	1	0	0	0	0	93.75
0	1	1	0	2	0	90.62
$VMs \in VMF_4$						
8	0	0	0	0	0	100
0	0	0	0	8	0	100
5	0	0	0	0	2	100
0	0	0	0	0	6	100
0	0	1	0	7	0	98.44
0	2	5	0	2	0	98.44
5	1	0	0	2	0	96.88
4	0	3	0	1	0	95.31
1	1	2	0	4	0	93.75
5	1	2	0	0	0	93.75
2	1	3	0	2	0	92.19
1	2	2	0	3	0	90.62
3	2	1	0	2	0	92.19
$VMs \in VMF_5$						
0	0	0	0	0	6	100
6	0	0	0	0	4	100
0	0	0	5	2	2	98.96
0	4	0	5	2	0	98.96
0	3	0	1	5	2	96.88
8	0	3	0	1	0	96.88
4	1	2	0	5	0	95.83
5	0	4	0	3	0	95.83
3	2	1	0	4	1	90.62
1	2	1	0	6	1	90.62
4	4	1	0	3	0	90.62
1	2	0	1	7	0	90.62
0	3	0	0	7	1	90.58

As the solution space is huge, we select z different combinations of map/reduce tasks that gives minimized resource wastage in each bin periodically for every C_t seconds. Because, scheduling decision must be near real-time as jobs are being executed. Constraints of map and reduce shrink the solution space that can be searched. As we are modifying fair scheduler, a fair share of resources among jobs must be ensured at every C_t. This is verified by calculating the amount vCPU and memory used by map/reduce tasks of different jobs at C_t with the overall resources available in the virtual cluster using Equation 4.6.

$$\forall_{j}, \frac{\sum_{r=1}^{p} AR_M_j^v + \sum_{s=1}^{q} AR_R_j^v}{\sum_{i=1}^{N} B_i^v} \times \frac{\sum_{r=1}^{p} AR_M_j^u + \sum_{s=1}^{q} AR_R_j^u}{\sum_{i=1}^{N} B_i^u} \leq \frac{\sum_{i=1}^{N} B_i^v}{|J|} \times \frac{\sum_{i=1}^{N} B_i^u}{|J|} \tag{4.6}$$

It is important to note that a fair share of resources for jobs is not assured in its whole lifetime, but it is ensured at every C_t when a job gets its opportunity satisfying the resource utilization. In addition, data locality is the primary constraint for map tasks to minimize the latency and ensured using Equation (4.7). In order to achieve this, we firstly get the block location information from namenode service and determine whether tasks can be launched in the respective bin ($Task_j^r \in B_i^{h,g}$) as three bins are available for each block. As RWS based block placement is used, it does not matter which bin is chosen for each block. Therefore, the bin that corresponds to maximum resource utilization is preferred.

$$\forall_{j,p}, AR_M_{i,j}^p \leftarrow Task_j^p \in TAR \tag{4.7}$$

Finally, to avoid overallocation of map/reduce tasks in $B_i^{h,g}$, the amount of resources planned for map and reduce tasks in each bin are summed up and compared with the amount of resources available in $B_i^{h,g}$ using Equation (4.8), in which vCPU and Memory are added separately and compared with the overall resources.

$$0 \leq \sum_{k=1}^{T_i} AR_M_k^v \vee AR_R_k^v \leq BR_i^v \wedge 0 \leq \sum_{k=1}^{T_i} AR_M_k^u \vee AR_R_k^u \leqslant BR_i^u \tag{4.8}$$

4.1.3 Packing map/reduce tasks using Ant Colony Optimization (ACO)

If there are hundreds of VMs and tens of MapReduce jobs, then finding all combinations of map/reduce tasks of different jobs in each VM is a time-consuming process. For instance, as shown in Figure 4.1, for two MapReduce jobs $< J_1,J_2 >$ to run in a bin, there are five possible combinations available. If there are 100 bins, then 5^{100} combinations are possible in the solution space, which is huge to find fair share among jobs, and we need a schedule in a few seconds. Moreover, when the number of jobs, the number of containers, and the number of bins increases, solution space goes exponentially high, which is not possible to evaluate in short-time for finalizing the schedule. Therefore,

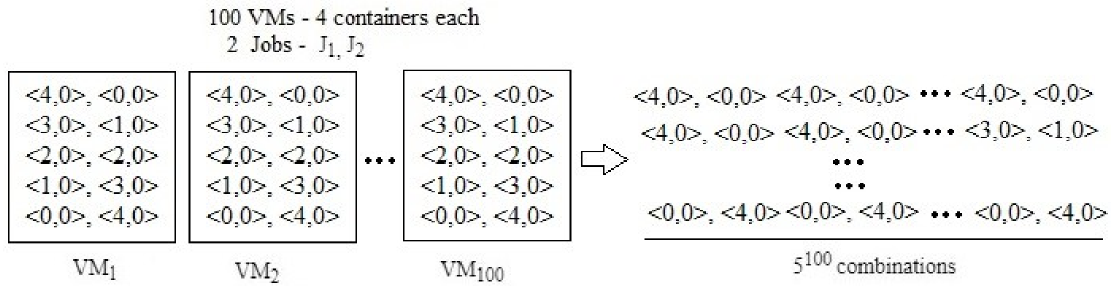


Figure 4.1 Number of map/reduce task combinations of different jobs

we used a meta-heuristic algorithm, ACO, which is well known for finding an optimal solution from huge discrete solution space. As given in Algorithm 6, we initially find the maximum number of map tasks of each job possible in each VM flavor, as listed in Table 4.3. For any solution space, we need to know the range within which we have to look for the solution. For instance, VMF_5 can execute 12, 12, 12, 6, 12, and 6 map tasks of J_1, J_2,..., and J_6 respectively. Now, in each bin that belongs to different flavor finds different combinations of map tasks from different jobs, subsequently, TAR_i is calculated, as given in Table 4.2. One possible combination of map tasks of each job in a bin that belongs to VMF_5 is (0, 0, 0, 5, 2, 2) and TAR_i is calculated. Similarly, we calculate the resource utilization of all possible combinations in the respective bin and choose combinations that has TAR_i over 90%. Table 4.2 displays the possible map tasks combination in a bin of each VM flavour when there are no tasks running previously. We prefer top 5 map task combinations of different jobs in each bin with data locality.

Table 4.3 Maximum number of map tasks for each job in each VM flavour

Job	VMF_1	VMF_2	VMF_3	VMF_4	VMF_5
1	1	2	4	8	12
2	1	2	4	8	12
3	1	2	4	8	12
4	0	1	2	4	6
5	1	2	4	8	12
6	0	1	2	4	6

For reduce tasks, we find a set of VMs running in a specific rack. Firstly, we find a rack that may transfer more map output data among all the racks in the CDC using Equation 4.9. As we cannot find the exact map output size in advance before all map tasks completed, we add up the map output stored in the in-memory buffer. After finding a rack that may cause more map output, we find a set of VMs in the rack that will produce more map output, in the same way, using Equation 4.10. After this, a right

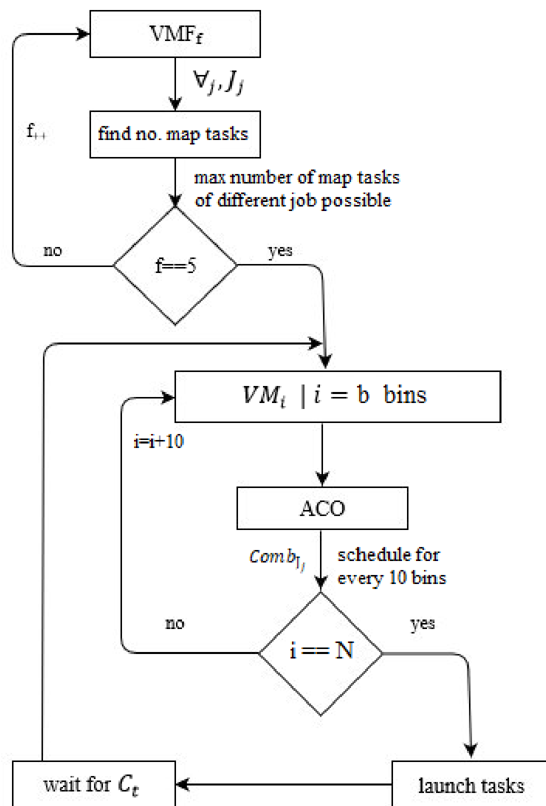


Figure 4.2 Finding map/reduce task combinations using ACO

Algorithm 6: Bin packing map/reduce tasks using ACO

1. Get the information on the workloads and bins.

2. Find the different combinations of map/reduce tasks of different jobs that can be run in each bin.
   For the map tasks

   (a) Find the maximum number of map tasks of each job possible in each VM falvor (Table 4.3) to set the limit for number of combinations.

   (b) Find all possible combinations of map tasks from all the jobs that are currently active and find TAR_i (Table 4.2).

   (c) Consider top z combinations of map tasks in each $B_i^{h,g}$ that belongs to a particular VM flavour ensuring data locality.

   For the reduce tasks.

   (a) Map output in racks

$$\forall g, MOR_g = \forall j, \frac{\sum_{i=1}^{N}(\sum_{r=1}^{p} TaskOutput_j^r \in B_i^{h,g})}{\sum_{g=1}^{X}(\sum_{i=1}^{N}(\sum_{r=1}^{p} TaskOutput_j^r \in B_i^{h,g}))} \quad (4.9)$$

   (b) Preferred rack to process the reduce task Pref_Rack=$max(MOR_g)$

   (c) Map output in VMs in Pref_Rack

$$\forall_i, MOV_i = \frac{\sum_{r=1}^{p} TaskOutput_j^r \in B_i^{h,PrefRrack}}{\sum_{i=1}^{N}\left(\sum_{r=1}^{p} TaskOutput_j^r \in B_i^{h,PrefRack}\right)} \quad (4.10)$$

   (d) Preferred VMs to process the reduce task Pref_VM=sort_des_order(MOV)

   (e) Consider the top 50% of bins from Pref_VM to launch reduce tasks.

   (f) Find the possible combinations $< map, reduce >$ tasks that belong to J_j in each bin (if early reduce enabled) and find TAR.

3. Apply ACO taking Table 4.2 as input for every 10 bins (Figure 4.2) to find the fair share among jobs and picking up the map/reduce task combinations that results TAR over 90%.

4. Collect schedule and commit tasks.

5. Check for unused resources every C_t, and repeat step 2 until all jobs are completed.

combination of $< map, reduce >$ is found for each job. If early reduce feature is enabled, then before all map tasks of a job is over, all reduce tasks are launched. Therefore, we need to include the reduce tasks also in finding the right combinations (as given in

ordered pair) that results to resource utilization. Mostly, 1000s of map tasks are executed for the huge dataset, so over 90% of the time only map tasks combination of different jobs are considered. This way, we minimize the inter-rack bandwidth consumption while launching reduce tasks. After finding possible bins to launch map and reduce tasks, as shown in Figure 4.2, we run ACO for every b bins to find the right combination of map/reduce tasks to maximize the individual bin resource utilization. For instance, $<4,0><0,0>$ from $B_1^{h,g}$, $<2,0><2,0>$ from $B_2^{h,g}$, $<1,0><3,0>$ from $B_3^{h,g}$, etc. In order to ensure a fair share of resources among jobs, we need to evaluate all 5^{100} combinations for 100 bins. Therefore, we breakdown the combinations by taking every b bins to find fair share, which will cause just 5^{10} combinations. This takes a few seconds to find the right combinations and schedule map/reduce tasks in real-time.

4.1.4 Fine Grained Data Locality Aware (FGDLA) job scheduling

Virtual network bandwidth consumption is always critical while sharing among a set of VMs. As MapReduce jobs consume an arbitrary amount of network bandwidth during the shuffle, it is important to minimize service cost and improve makespan. As already mentioned, MLPNC was introduced to minimize the number of intermediate records for a job. While running a batch of jobs, map tasks of different jobs are executed arbitrarily depending upon data locality. Therefore, it becomes challenging to find a set of map tasks that belong to the same job in a node in a given time, to run a single combiner common for them. To minimize virtual network bandwidth consumption further, we introduce FGDLA by extending MLPNC for a batch of jobs. MLPNC is fruitful only when a greater number of map tasks of the same job is executed in a node so that the number of times a combiner invoked is minimized. If there are four map tasks from different jobs running in a bin, there has to be four MLPNC running because each job has its combiner functionality.

Therefore, each MLPNC reserves some memory and runs as a separate thread. This adds overhead, and there is no much reduction in map output data size. However, if there are four map tasks of the same job running in a bin, then it is enough to run its respective MLPNC. While running a batch of jobs, it is tricky to ensure all map tasks are from the same job to ensure only one MLPNC is running. So, while executing a

batch of jobs, FGDLA schedules a set of map tasks of a job to exploit MLPNC. For instance, as shown in Figure 4.3, consider Rack1 containing PMs (PM1, PM2), four VMs (VM1, VM2, VM3, VM4) running in different PMs, and two MapReduce jobs (job1, job2) with 16 blocks and 8 reduce tasks each. Each VM contains a different number of containers, for instance VM1, VM2, VM3, VM4 contains (C1, C2), (C1, C2, C3), (C1, C2, C3, C4), and (C1, C2) containers respectively. Blocks of job1 are denoted as A1...A16 and for job2 as B1...B16. The same representation is used for map task execution, as shown in Figure 4.4. Reduce tasks of job1 are denoted as AR1...AR8, and for job2 as BR1...BR8. Non-local execution for map tasks is circled. We discuss FGDLA for a batch of two jobs scheduled with FCFS, and Fair scheduler. *x* and *y* axis in the graph (Figure 4.4) is a task commit sequence, and VMs chosen for tasks respectively. In every commit sequence, a set of tasks of different jobs is scheduled for execution. In FCFS, jobs are scheduled based on arrival time. A job that comes first gets complete resources available in the cluster. So, job1 is executed in the first commit sequence. Job2 starts executing in the second commit sequence once map tasks of job1 completed. Therefore, there is very less chance of non-local execution as map tasks

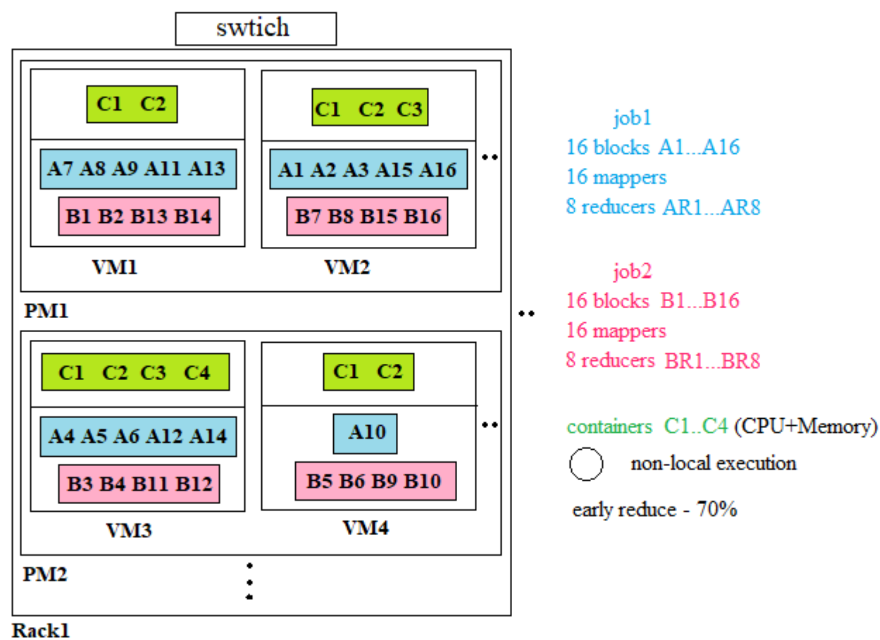


Figure 4.3 Example using FGDLA

		1	2	3	4	5	6
FCFS	VM1	A1 A2	A15 A16	B7 B8	B15 B16	BR1 BR2	BR1 BR2
	VM2	A7 A8 A9	A11 A13 AR1	B1 AR7 AR1	B2 B13 B14	BR3 BR4 BR7	BR3 BR4 BR7
	VM3	A4 A5 A6 A12	A14 AR2 AR3 AR4	AR8 AR4 AR3 AR2	B3 B4 B11 B12	BR5 BR6 BR8	BR5 BR6 BR8
	VM4	A10 A3	AR5 AR6	AR5 AR6	B5 B6	B9 B10	
FAIR	VM1	A1 B7	A2 B8	A15 B15	AR1 BR1	AR1 BR7	
	VM2	A7 B1 B2	A8 B13 B14	A9 A16 B16	A13 BR2 BR3	AR3 BR8	
	VM3	A4 A5 B3 B4	A6 A12 B11 B12	A14 A11 B10 BR5	AR2 AR4 BR4 BR5	AR2 AR4	
	VM4	A10 B5	A3 B6	A16 B9	AR5 BR6	AR5	
FGDLA	VM1	A1 A2	B7 B8	A15 A16	AR6 AR4	BR4	
	VM2	A7 A8 A9	B1 B2 B3	A11 A13 B14	AR5 AR7 BR3	BR5	
	VM3	A4 A5 A6 A12	B3 B4 B11 B12	A14 A3 AR1 BR1	AR2 AR3 AR1 BR1	BR6 BR8	
	VM4	A10 B5	B6 B9	B10 BR2	AR8 BR2	BR7	

commit sequence

Figure 4.4 FCFS vs FAIR vs FGDLA

are scheduled when a node has free container. However, makespan is affected because job2 comes into picture only after all map/reduce tasks allocated. In fair scheduling, jobs are given an equal share of resources from the cluster. By doing so, small jobs can be finished early, and makespan also improved a little bit. Since map tasks of all jobs are executed simultaneously, there may not be chances to exploit MLPNC. The number of non-local execution also could be higher (circled) as jobs are guaranteed with an equal share of resources but not with the chance of data locality. In contrast, FGDLA takes a different approach using time-sharing (round robin) fashion to provide all available resources to each job but in different commit sequence. Therefore, both jobs are executed in alternative cycle of commit sequence as shown in Figure 4.4. By doing so, the number of non-local execution can be minimized as in FCFS, further providing a fair chance for each job. The advantage is, the shortest job need not wait for a long time if a long job gets its chance in the first commit sequence. Additionally, in every commit sequence, it is ensured that the maximum number of map tasks of a job are executed together. It helps to extend MLPNC to minimize the number of intermediate records further. Since FGDLA is a combination of FCFS and time-sharing strategy, the number of non-local execution also is largely minimized. Algorithm 7

exhibits the procedures in short. Once FGDLA applied for a batch of jobs, default combiner function and MLPNC can be applied. More specifically, MLPNC is extended to minimize the number of intermediate records, thereby minimizing the makespan.

Algorithm 7: Fine Grained Data Locality Aware scheduler

1. Find the bins where map tasks of each job are executed.
2. Find maximum number of map tasks of each job possible in $B_i^{h,g}$ at every C_t.
3. Launch J_j in time-sharing fashion at every C_t.
4. If not all map tasks of a job can be placed, next job is given an opportunity in fine-grained fashion sharing time.
5. Apply MLPNC.

4.2 Results and Analysis

4.2.1 Bin packing map/reduce tasks using ACO

4.2.1.1 Experimental setup

In this section, we discussed the efficiency and effectiveness of our proposed model over classical fair scheduler in Hadoop MapReduce v2 as there is no bin packing model for map/reduce tasks to the best of our knowledge. We implemented our MapReduce simulator [69] running in a server computer with 12 core (hyper-threaded), 32 GB memory. Along with the information given in Figure 1.13, we additionally consider the following parameters for VM and workloads in our experiment.

- PMs in CDC are heterogeneous.
- Number of VMs (bins) is fixed and deployed across the racks in CDC.
- There are five different flavors of VMs and 20 VMs in each flavor.
- No interference is assumed by co-located VMs.
- Capacity $< vCPU, Memory, HDD >$ of VMs is heterogeneous.
- Dimensions for bin packing are $< vCPU, Memory >$
- Constraints: data locality for map tasks, fair sharing among jobs at every C_t.
- Number of workloads (jobs) are fixed (batch processing).
- Workloads are wordcount, wordmean, word standard deviation, kmean, sort, join.
- Workloads demand heterogeneous containers (varying size of $< vCPU, Memory >$).

- Size of dataset (in HDFS) considered for these six workloads is 128 GB, 64 GB, 256 GB, 192 GB, 76.8 GB, 153.6 GB respectively and 870.4 GB in total.
- HDFS block size is 128 MB with replication factor 3.
- Number of map tasks (one map task for each block) possible for each job is 1000, 500, 2000, 1500, 600, and 1200, respectively.
- Number of reduce tasks of each job is: 20, 15, 50, 10, 0, and 5, respectively.

We developed two modules along with the basic MapReduce functionalities to assist fair scheduler. The first module overrides the default block placement scheme in HDFS to place data blocks based on RWS. The second module assists fair scheduler to place map/reduce tasks with the right combination to maximize resource utilization, thus minimizing job latency and makespan consequently. We compare and contrast the results of our model with classical fair scheduler based on latency, makespan, number of non-local execution, and average number of unused vCPU and memory.

4.2.1.2 Results and Analysis

We used general ACO algorithm with pheromone decay factor 0.1 to get the solution quicker, and pheromone value for each path is 1. If pheromone decay factor is high, then the exploration time is high, which takes a high number of iterations to find an optimal solution. We fixed C_t to be 10 seconds as average map task latency in the batch is 10 seconds. This is observed as the batch of jobs are being periodically executed, however, this can be modified. Moreover, it considers top 5 combinations from each bin that results TAR over 90% to find fair share of resources among all the jobs at every C_t. Every ten bins are passed to ACO to find a good schedule as the number of combinations to evaluate is very less for ACO. We compared and contrasted three different cases in this regard: Case 1.Classical fair scheduler with default block placement [22], Case 2.Classical fair scheudler with RWS block placement scheme, and Case 3.ACO based bin packing map/reduce tasks with RWS block placement scheme.

Figure 4.5 shows the latency of each workload for the three cases. Case 2 minimized 7%-21% of latency for each workload. The latency of wordcount, wordmean, word standard deviation, kmean, sort, and join jobs minimized up to 17.6%, 15.2%, 9%, 7%,

-3%, and 20.7% respectively. Sort job latency slightly increased due to more number of non-local execution for map tasks. Similarly, wordcount, wordmean, word standard deviation, kmean, sort, and join jobs latency slashed up to 52.5%, 33.7%, 59.4%, 60.4%, -0.09%, and 68.8% respectively using Case 3 over Case 1. Case 3 also minimized job latency up to 42.3%, 21.8%, 55.2%, 57.2%, -0.06%, and 60.8% respectively over Case 2. Minimization of latency is possible for two reasons: Firstly, placing data blocks based on the number of containers (vCPU) possible in each VM flavor, and finiding a different combination of map/reduce tasks in VMs of different flavors. In average, Case 2 and Case 3 achieved up to 11.1%, and 44.1% improvement in latency over Case 1 while Case 3 achieved 38.6% improvement in latency over Case 2. Figure 4.6 exhibits

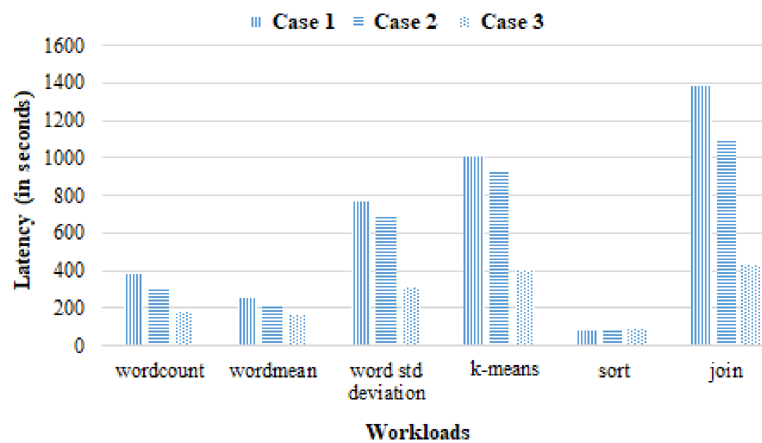


Figure 4.5 Latency of jobs using different schedulers

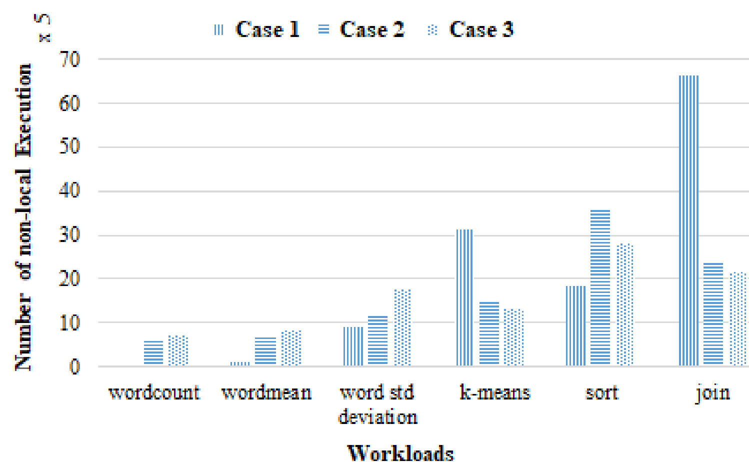


Figure 4.6 Number of non-local executions

the number of non-local execution for map tasks of different jobs under three different cases. Our another claim concerning number non-local execution is, Case 2 and Case 3 minimized non-local execution up to 21.1% and 24.3% respectively over Case 1. Case 3 did not show much improvement in minimizing the number of non-local execution compared to Case 2. It is because, in order to improve resource utilization using a bin packing model, sometimes it is required to compromise with the non-local executions. Therefore, Case 3 just improved 0.04% in minimizing the number of non-local execution over Case 2. Being more resource-aware helps to place the right combination of map/reduce tasks in each VM using ACO. As shown in Table 4.2, any one of the combinations in each bin that belongs to a particular VM flavor is considered to pack tasks. Therefore, it is possible to schedule tasks without wasting much resources.

As Case 3 scheduler monitors the resource availability every C_t, ACO comes up with different possible combinations to improve resource utilization. As thousands of map tasks are scheduled, most of the combination could be map tasks of different jobs. Therefore, we discuss the results for bin packing concentrating map task combinations of different jobs. Along with the improvements in latency, makespan also has largely improved with our model, as shown in Figure 4.7. Case 2 and Case 3 improved makespan up to 19.6% and 57.9% respectively, over Case 1. Especially, Case 3 improved makespan up to 47.7% over Case 2. This is possible because ACO finds different combinations in each bin that gives over 90% resource utilization. Our primary motivation behind using RWS based block scheme is bin packing map/reduce tasks by loading more blocks on to the larger bins. This will increase the number of data local execution. Figure 4.8 and Figure 4.9 illustrate the utilization of vCPU and memory using three cases at every 10 seconds. As ACO finds the right combination of tasks in each bin, resources are utilized maximum at any point of time. So, the amount of unused resources such as vCPU and memory is minimized compared to the other two cases. At times, in order to keep resources busy, some non-local execution is performed, which indirectly affects job latency slightly. However, Case 3 has largely minimized the resource wastage compared to the other two cases as shown in Figure 4.10.

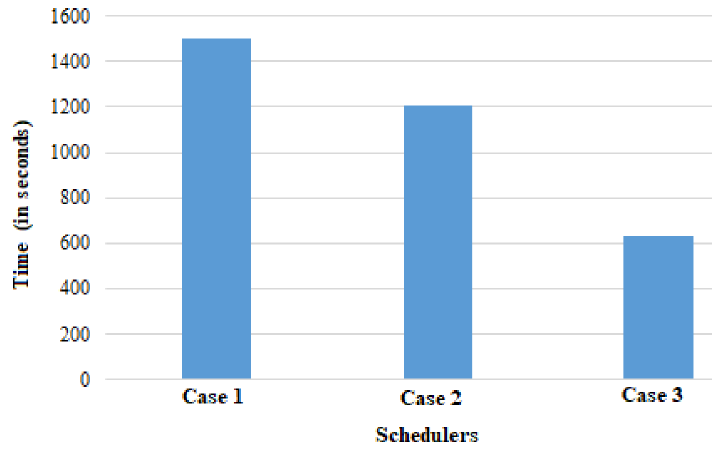


Figure 4.7 Makespan

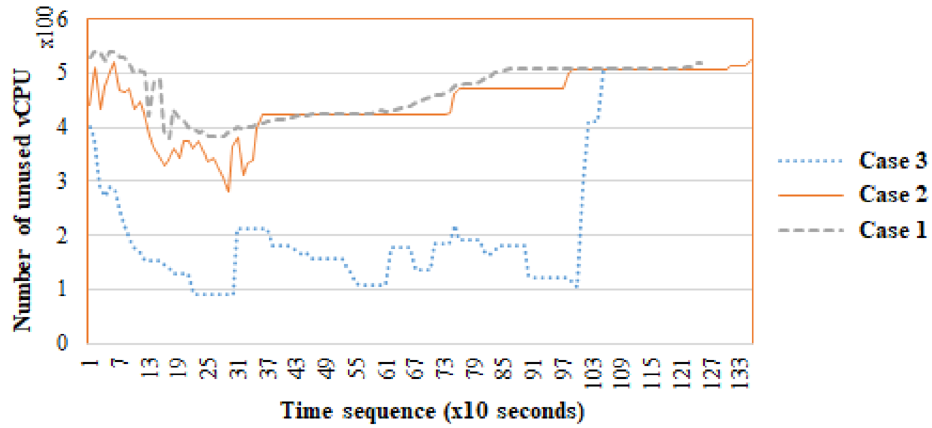


Figure 4.8 Utilization of the vCPU

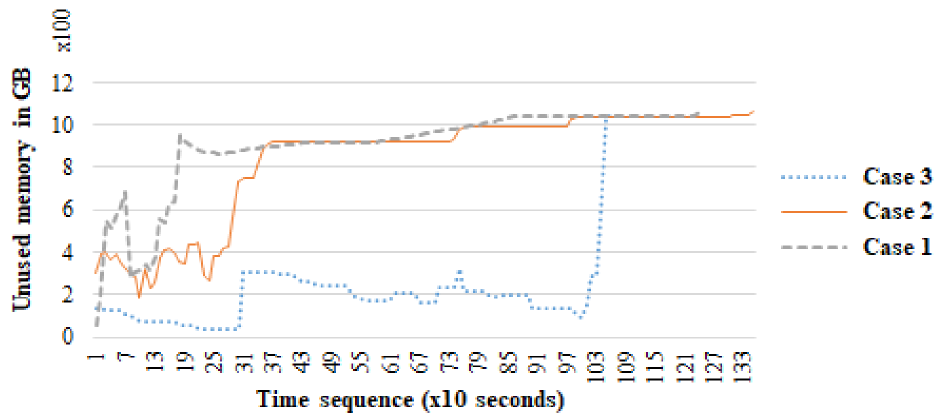


Figure 4.9 Utilization of the memory

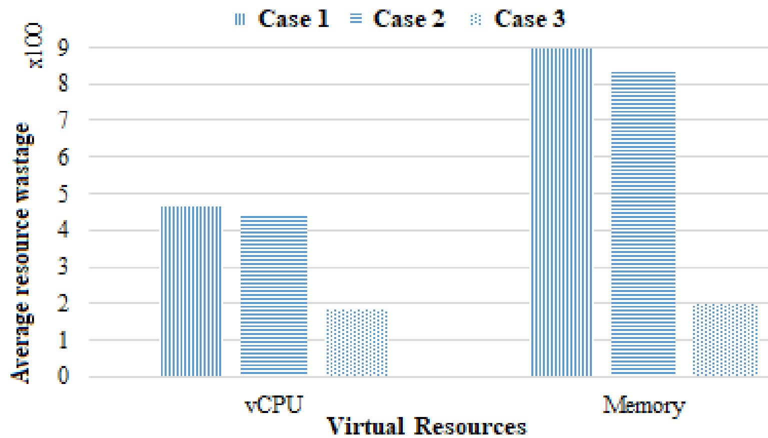


Figure 4.10 Average resource wastage of different schedulers

Case 2 also minimized the idle resources (vCPU and memory) up to 3% and 6% respectively, over Case 1. However, Case 3 has minimized the idle resources (vCPU and memory) up to 60.8% and 77.6% respectively, over Case 1. Case 3 has also improved up to 59.3% and 75.9% for vCPU and memory, respectively compared to Case 2.

4.2.2 Fine-Grained Data Locality-Aware scheduler (FGDLA)

As already discussed in Section 3.2.2, MLPNC considerably minimized the number of intermediate records for a single job. To generalize MLPNC for a batch of jobs, we have to make sure more number of map tasks of a job to run at any time. FGDLA achieves this idea and extends MLPNC to minimize the makespan. In this section, we simulated and evaluated the performance of FGDLA with MLPNC by comparing with classical schedulers (FCFS and fair scheduler [22]), FGDLA, FGDLA with default combiner for a batch of jobs in Hadoop MapReduce v2. We implemented our MapReduce simulator [69] for these schedulers and used a server computer with 12 core hyper-threaded, 32 GB memory for simulation. We set the following parameters for VMs and workloads in our experiment.

- Physical servers in CDC are heterogeneous.
- We used 100 VMs and deployed across the racks in CDC.
- No interference is considered among VMs.

- Workloads are wordcount (J_1), wordmean (J_2), wordmedian (J_3), and kmean (J_4) in the sizes of 128 GB, 64 GB, 256 GB, 192 GB respectively and 640 GB in total.
- Workload description:
  - Wordcount job counts the occurrences of each word given in the files.
  - Wordmean job counts the average length of the words in the given files.
  - Wordmedian job counts the median length of the words in the given files.
  - Kmean job finds the cluster of similar elements from the given numeric data.
- HDFS block size is 128 MB with replication factor 3.
- Workloads demand heterogeneous containers (varying size of <vCPU, Memory>) as shown in Figure 4.11.
- The number of map tasks (one map task for each block) possible for each job is 1000, 500, 2000, and 1500 respectively.
- A number of reduce tasks for each job is: 20, 15, 50, and 10, respectively.

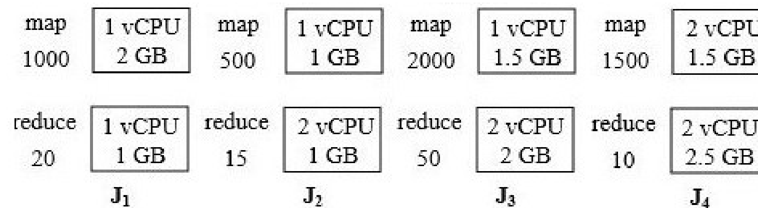


Figure 4.11 Resource requirements of each job

We have considered these workloads as it can apply combiner function to minimize the size of intermediate data. Combiner function can be either reduce function itself or user-defined function. To use reduce function itself as combiner function, it should satisfy commutative and associative properties on the map output records. Users also can define custom combiner function that can minimize shuffle data. We have compared and contrasted our approach with other schedulers based on the parameters such as job latency, makespan, number of non-local execution, size of shuffle data, and finally, the amount of resources unused. A major motivation of FGLDA to extend MLPNC is to minimize the amount of intermediate data transferred in the shuffle phase, and job latency, thereby minimizing the makespan for a batch of jobs.

This is shown in Figure 4.12, in which we compared the latency of J_1, J_2, J_3, and J_4 jobs. Initially, we compared FGDLA with FCFS and Fair schedulers. Then, we compared FGDLA against FGDLA with the default combiner and FGDLA with MLPNC. As shown in Figure 4.12, FGDLA improved latency of J_1, J_2, J_3, and J_4 jobs by 43.6%, 17.1%, 20.7%, and 46.8% compared to Fair scheduler and by 43.5%, 18%, 6%, and 57.5% compared to FCFS scheduler. This large minimization is possible by minimizing the number of non-local execution. FGDLA minimized the number of non-local execution up to 31.1%, and 14.9% over FCFS and Fair scheduler as shown in Figure 4.13. As

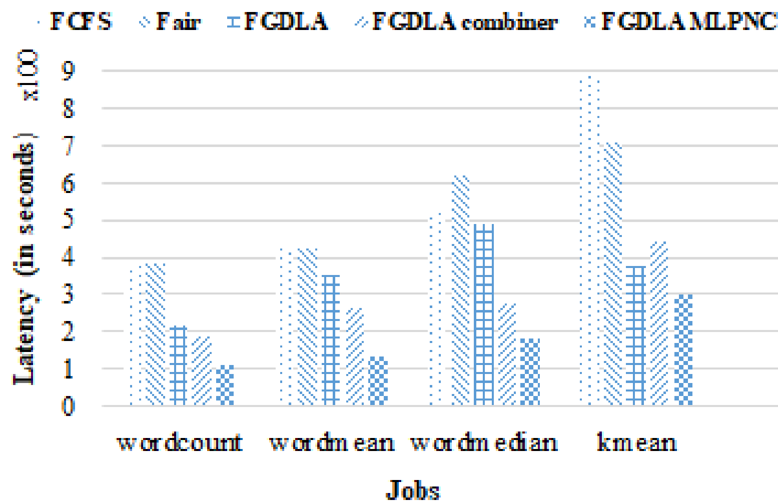


Figure 4.12 Latency of jobs

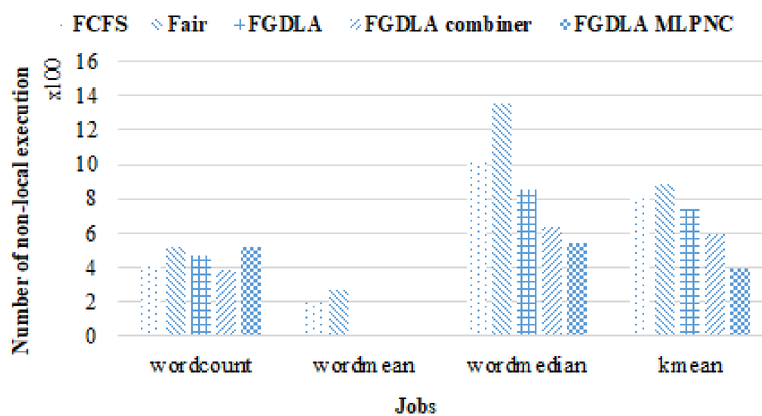


Figure 4.13 Number of non-local executions

FGDLA outperformed the classical schedulers for a batch of jobs, we evaluate FGDLA with default combiner and FGDLA extending MLPNC. As a result, FGDLA with default combiner minimized the job latency further up to 12.5%, 24.2%, 43.3%, and -10% respectively over FGDLA. Here, the kmeans algorithm performs more computation in combiner function, as it takes an extra 10% latency when compared to FGDLA. After extending MLPNC for FGDLA, it further minimized the job latency by 41.2%, 49.4%, 35.6%, and 32.7% respectively when compared to FGDLA with default combiner. Similarly, the number of non-local execution is also minimized up to 9% when compared to FGDLA with default combiner.

The major claim of this work is to minimize the size of intermediate data in the shuffle phase. FCFS, Fair, and FGDLA scheduler transfer 356.3 GB of intermediate data in total, as shown in Figure 4.14. After applying default combiner in FGDLA, size of intermediate reduced to 48 GB in total. However, extending MLPNC with FGDLA minimized the size of shuffle data further by 62.1%. This is mainly because the FGDLA algorithm allows each job to time share in every commit sequence. Therefore, intermediate data is written in the in-memory cache for each job entirely. Scheduling many map tasks of the same job in a node minimizes the number of non-local execution as already explained. These two factors together helped to minimize the makespan for FGDLA up to 21.4%, and 30% compared to FCFS and Fair scheduler, as shown in Figure 4.15. Similarly, FGDLA with default combiner reduced makespan by 29.4% compared to

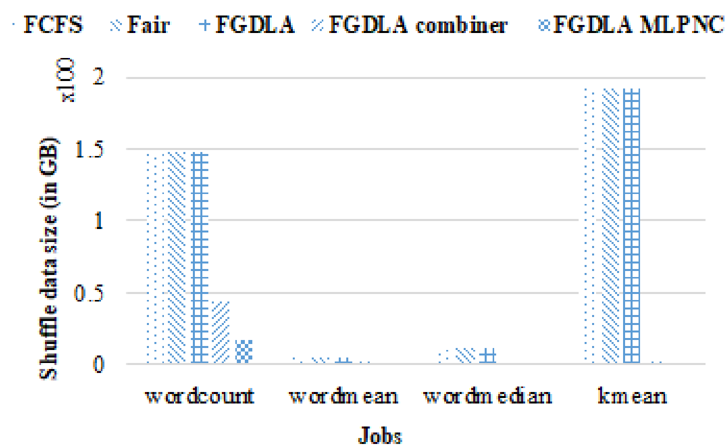


Figure 4.14 Size of shuffle data

FGDLA. Finally, FGDLA with MLPNC improved makespan over 32.4% compared to FGDLA with default combiner.

FGDLA also improved the resource utilization of hired virtual resources. Cloud users expect all the hired resources to be utilized in a given time. As shown in Figure 4.16, FGDLA minimized unused vCPU up to 2%, and 8% compared to FCFS and Fair scheduler on average. 21.3% improvement was observed by using FGDLA with default combiner over FGDLA. By extending MLPNC with FGDLA, unused vCPU is minimized by 18.2% further over FGDLA with default combiner. Similarly, FGDLA minimized unused memory up to 9%, and 12.7% compared to FCFS and Fair scheduler.

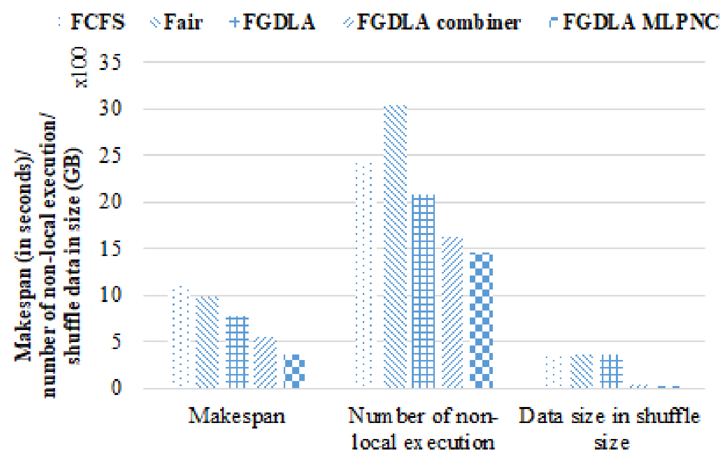


Figure 4.15 Makespan, number non-local executions, shuffle data size of each job

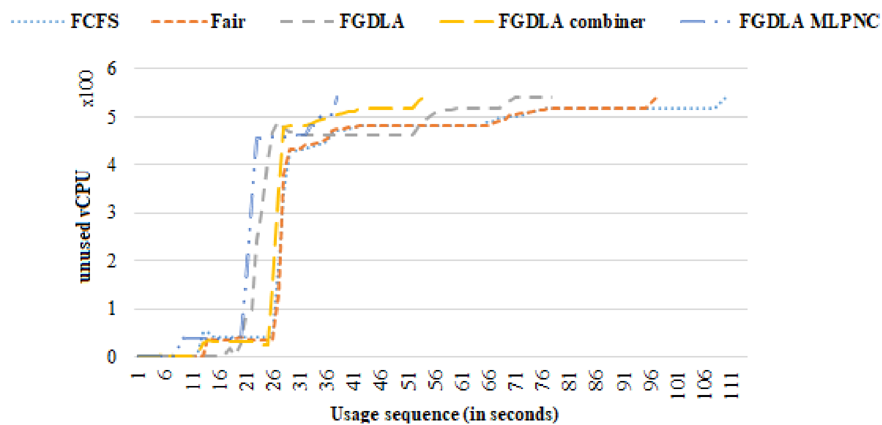


Figure 4.16 Unused number of vCPUs during execution

FGDLA with MLPNC largely minimized the unused memory by 28.7% compared to FGDLA with default combiner, as shown in Figure 4.17. As combiner is split from map tasks and run as a separate thread, more map tasks can be scheduled in a short time.

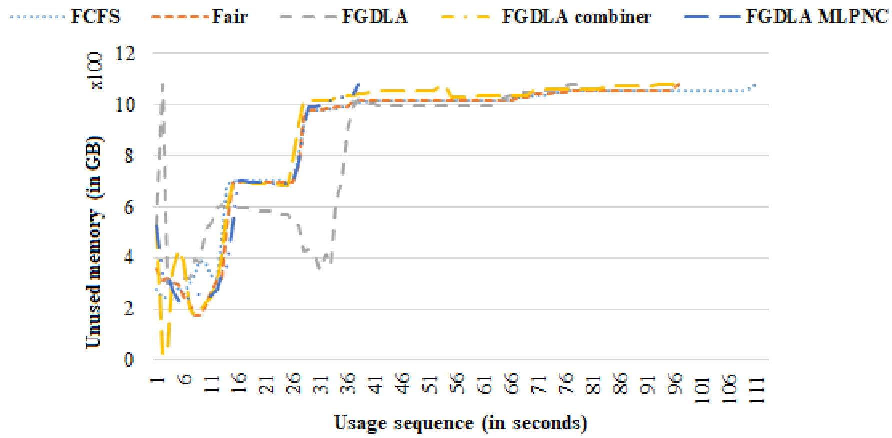


Figure 4.17 Unused memory during execution

4.3 Summary

Hadoop MapReduce on cloud is increasingly being used. Although resources are scalable on demand, it is doubtful whether all the hired resources are utilized entirely or not. As MapReduce jobs and VM capacities are becoming heterogeneous, it is challenging to schedule map/reduce tasks in order to improve makespan and resource utilization. Considering this, we proposed RWS to place data blocks of different workloads based on the capacity of VM to minimize job latency. To improve resource utilization, we introduced a constrained 2-dimensional bin packing model to find the right combination of map/reduce tasks using ACO to improve resource utilization. As we expected, our proposed model improved makespan and resource utilization up to 57.9% and 59.3% over classical MapReduce fair scheduler. It is observed that 26%-70% of MapReduce job latency is due to the shuffle phase affecting the overall makespan. Therefore, we proposed FGDLA to form a set of map tasks of a same job, thereby MLPNC can be extended for a batch of jobs. FGDLA with MLPNC minimized the amount of intermediate data up to 62.1% when compared to FGDLA with combiner. As a result, makespan also was improved by 32.4% over FGDLA with combiner.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Hadoop MapReduce on cloud is increasingly being used. The primary concern for Hadoop users is improving job latency, makespan, and resource utilization. However, Hadoop in virtual environment faces many challenges as Hadoop VMs are spread across racks in CDC. One of the significant problems is the heterogeneous performance of VMs due to hardware heterogeneity and co-located VM's interference. It primarily impacts job latency, makespan, and resource utilization. So, we proposed DRMJS to improve job latency and resource utilization in a virtual cluster. DRMJS improved overall job latency, makespan, and resource utilization up to 30%, 28%, and 60%, respectively, on average compared to existing MapReduce schedulers. To minimize job latency further, we introduced MLPNC to minimize the number of intermediate records, there by minimizing the job latency. MLPNC outperformed PNC up to 33% reduction in number of shuffled records, and up to 32% reduction in average job latency. We also placed reduce tasks based on its input size by exploiting heterogeneous performance after MLPNC. Results claimed that 28%-41% of reduce task latency improvement is possible by exploiting performance heterogeneity. Although resources are scalable on demand, it is doubtful whether all the hired resources are utilized entirely or not. As MapReduce jobs and VM capacities are becoming heterogeneous, it is challenging to schedule map/reduce tasks in order to improve makespan for a batch of jobs and resource utilization. Considering this, we introduced RWS to place data blocks of different workloads based on the capacity of VM. To improve resource utilization, we introduced a constrained 2-dimensional bin packing model to find the right combina-

tion of map/reduce tasks using ACO to minimize the resource wastage in each bin. As we expected, our proposed model improved makespan and resource utilization up to 57.9% and 59.3% over classical MapReduce fair scheduler. Minimizing MapReduce job latency is very important to minimize makespan while executing a batch of jobs. Therefore, we proposed FGDLA to form a set of map tasks of a same job, thereby MLPNC can be extended for a batch of jobs. FGDLA with MLPNC minimized the amount of intermediate data up to 62.1% when compared to FGDLA with combiner. As a result, makespan also was improved by 32.4% over FGDLA with combiner.

5.2 Future Work

There are ample opportunities to improve job latency, makespan, and resource utilization further for MapReduce job and task scheduling in a virtualized environment. An interesting research question is, can we bin pack right mix of CPU-intensive and IO intensive tasks of different jobs to improve resource utilization?

Chapter 6

Appendix

6.1 Course Work

S. No.	Course Code	Course Name	credits	Grade
1	IT701	Advanced Database Systems	4	AA
2	HU800	Research Methodology	2	S
3	MA712	Optimization Techniques and Random Process	4	CC
4	CS867	Data Science	3	AA
5	IT806	Distributed Computing Systems	3	AB
		Earned credits	**16**	
		CGPA	**8.64**	

6.2 Work Timeline

Target	Dec 2015	May 2016	Jan 2017	Feb 2018	Mar 2019	Aug 2019	Sep 2019
Course work	√	√					
Literature survey	√	√	√	√	√	√	√
Research proposal			√				
Progress seminar - I				√			
Progress seminar - II					√		
Pre-synopsis						√	
Thesis submission							√

6.3 List of Publications

6.3.1 International Journals

1. Rathinaraja Jeyaraj, Ananthanarayana V S, Anand Paul, "Fine-grained data-locality aware MapReduce job scheduler in a virtualized environment," accepted in Journal of Ambient Intelligence and Humanized Computing (Springer), **2020**. (SCIE and Scopus, IF: 1.9)

2. Rathinaraja Jeyaraj, Ananthanarayana V S, Anand Paul, "Improving performance of MapReduce Scheduler for Heterogeneous Workloads in a Heterogeneous Environment," Concurrency and Computation: Practice and Experience (Wiley), **2019**. (SCIE and Scopus, IF: 1.1)

3. Rathinaraja Jeyaraj, Ananthanarayana V S, Anand Paul, "Dynamic Ranking based MapReduce Job Scheduler to Exploit Heterogeneous Performance on Virtualized Environment," Journal of Supercomputing (Springer), pp. 1-30, **2019**. (SCI, IF: 2.15)

6.3.2 International Conferences

1. Rathinaraja Jeyaraj, Ananthanarayana V S, Anand Paul, "MapReduce Scheduler to Minimize the Size of Intermediate Data in Shuffle Phase," 18th International Conference on Computer and Information Science (ICIS), June 17-19, **2019**, Beijing, China. (Core C, SCI).

2. Rathinaraja Jeyaraj, Ananthanarayana V S, "Dynamic Performance Aware Reduce Task Scheduling in MapReduce on Virtualized Environment," 16th IEEE/ACIS International Conference on Software Engineering Research, Management and Applications (SERA), June 13-15, **2018**, Kunming, China. (Core B3, EI, DBLP)

3. Rathinaraja Jeyaraj, Ananthanarayana V S, "Multi-Level Per Node Combiner (MLPNC) to Minimize MapReduce Job Latency on Virtualized Environment," 33rd ACM Symposium on Applied Computing (SAC), April 9-13, **2018**, Pau, France, pp. 167-174. (Core A1).

6.4 References

[1] I.A.T.Hashem, I.Yaqoob, N.B.Anuar, S.Mokhtar, A.Gani, and S.Ullah Khan, "The rise of big data on cloud computing: Review and open research issues," Information Systems, vol. 47, pp. 98-115, 2015.

[2] Yanfei Guo, Jia Rao, Changjun Jiang, "Moving Hadoop into the Cloud with Flexible Slot Management and Speculative Execution," IEEE Transactions on Parallel and Distributed Systems, Vol. 28, No. 3, pp. 798-812, 2017.

[3] D.H.Shin, "Demystifying big data: Anatomy of big data developmental process," Telecommunications Policy, vol. 40, no. 9, pp. 837-854, 2016.

[4] J.Dean and S.Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Sixth Symposium on Operating System Design and Implementation, pp. 137-149, 2004.

[5] R.Boutaba, L.Cheng, and Q.Zhang, "On Cloud computational models and the heterogeneity challenge," Journal of Internet Services and Applications, vol. 3, no. 1, pp. 77-86, 2012.

[6] Zhuoyao Zhang, Ludmila Cherkasova, Boon Thau Loo, "Exploiting Cloud Heterogeneity to Optimize Performance and Cost of MapReduce Processing," ACM SIGMETRICS Performance Evaluation Review archive, Volume 42, Issue 4, Pages 38-50, 2015.

[7] https://engineering.purdue.edu/ puma/datasets.htm, last accessed online July 2019.

[8] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, Ion Stoica., "Managing data transfers in computer clusters with orchestra," ACM SIGCOMM Computer Communication Review, Vol. 41 (4), pp. 98-109, 2011.

[9] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, and P. Valduriez, "FP-Hadoop: Efficient processing of skewed MapReduce jobs," Information Systems, vol. 60, pp. 69-84, 2016.

[10] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: Lightweight Data Skew Mitigation in MapReduce," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 9, pp. 2520-2533, 2015.

[11] W.H. Lee, H.G. Jun, and H.J. Kim, "Hadoop Mapreduce Performance Enhancement Using In-Node Combiners," International Journal of Computer Science and Information Technology, vol. 7, no. 5, pp. 1-17, 2015.

[12] Y.Guo, J.Rao, D.Cheng, and S.Member, "iShuffle: Improving Hadoop Performance with Shuffle-on-Write," IEEE Transactions on Parallel and Distributed Systems, vol. 9219, no. 2, 2016.

[13] Y.Wang, C.Xu, X.Li, and W.Yu, "JVM-bypass for efficient Hadoop shuffling," IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 569-578, 2013.

[14] H.Ke, P.Li, S.Guo, and I.Stojmenovic, "Aggregation on the fly: Reducing traffic for big data in the cloud," IEEE Network, vol. 29, no. 5, pp. 17-23, 2015.

[15] D.Guo, J.Xie, X.Zhou, X.Zhu, W.Wei, and X.Luo, "Exploiting efficient and scalable shuffle transfers in future data center networks," IEEE Transactions on Parallel and Distributed Systems, vol. 26, no. 4, pp. 997-1009, 2015.

[16] W.Shi, Y. Wang, J.P.Corriveau, B.Niu, W.L.Croft, and M.Peng, "Smart Shuffling in MapReduce: A Solution to Balance Network Traffic and Workloads," IEEE/ACM 8th International Conference on Utility and Cloud Computing, pp. 35-44, 2015.

[17] P.Costa, A.Donnelly, A.Rowstron, and G.O.Shea, "Camdoop: Exploiting In-network Aggregation for Big Data Applications," 9th USENIX Symposium on Networked Systems Design and Implementation, pp. 1-14, 2012.

[18] F.Liang and F.C.M.Lau, BAShuffler: Maximizing Network Bandwidth Utilization in the Shuffle of YARN, 25th ACM International Symposium on High Performance Parallel and Distributed Computing, pp. 281-284, 2016.

[19] Y.Yao, J.Tai, B.Sheng, and N.Mi, “LsPS: A Job Size-Based Scheduler for Efficient Task Assignments in Hadoop,” IEEE Transactions on Cloud Computing, vol. 3, no. 4, pp. 411-424, 2015.

[20] R.Y.Ming-Chang Lee, Jia-Chun Lin, “Hybrid Job-Driven Scheduling in Virtual MapReduce Cluster,” IEEE transactions on parallel and distributed systems, vol. 27, pp. 1687-1699, 2016.

[21] A.Verma, L.Cherkasova, “Orchestrating an Ensemble of MapReduce Jobs for Minimizing Their Makespan,” IEEE Transactions on Dependable and Secure Computing, Vol. 10, Issue 5, 2013.

[22] “Hadoop MapReduce Fair Scheduler,” Available: https://hadoop.apache.org/docs /r2.7.4/hadoop-yarn/hadoop-yarn-site/FairScheduler.html. [Last accessed: July, 2019]

[23] “Hadoop MapReduce Capacity Scheduler,” Available: https://hadoop.apache.org/ docs/r2.7.4/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html. [Last accessed: July, 2019]

[24] W.Hu et al., “Multiple-job optimization in mapreduce for heterogeneous workloads,” Sixth International Conference on Semantics, Knowledge and Grids, pp. 135-140, 2010.

[25] D.Cheng, J.Rao, Y.Guo, C.Jiang, and X.Zhou, “Improving Performance of Heterogeneous MapReduce Clusters with Adaptive Task Tuning,” IEEE transactions on parallel and distributed systems, vol. 28, no. 3, pp. 774-786, 2017.

[26] S.Tang, B.S.Lee, and B.He, “Dynamic Job Ordering and Slot Configurations for MapReduce Workloads,” IEEE Transactions on Services Computing, vol. 9, no. 1, pp. 4-17, 2016.

[27] Y.Yao, J.Wang, B.Sheng, C.C.Tan, and N.Mi, “Self-Adjusting Slot Configurations for Homogeneous and Heterogeneous Hadoop Clusters,” IEEE Transactions on Cloud Computing, vol. 5, no. 2, pp. 344-357, 2017.

[28] Z.Ming-Chang Lee, Jia-Chun Lin, and Ramin Yahyapour, “Hybrid Job-Driven Scheduling for Virtual MapReduce Clusters,” IEEE transactions on parallel and distributed systems, vol. 27, no. 6, 2016.

[29] Zhendong Bei, Zhibin Yu, Huiling Zhang, Wen Xiong, Chengzhong Xu, Lieven Eeckhout, Shengzhong, “FengRFHOC: A Random-Forest Approach to Auto-Tuning Hadoop’s Configuration,” IEEE transactions on parallel and distributed systems, vol. 27, no. 5, 2016.

[30] Yiduo Mei, Ling Liu, Xing Pu, Sankaran Sivathanu, “Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud,” IEEE 3rd International Conference on Cloud Computing, pp. 59-66, 2010.

[31] Ron C. Chiang, H.Howie Huang, “TRACON: Interference-Aware Scheduling for Data-Intensive Applications in Virtualized Environments,” IEEE transactions on parallel and distributed systems, Vol. 25, No. 5, pp.1349-1358, 2014.

[32] Xiangping Bu, Jia Rao, Cheng-Zhong Xu, “Interference and Locality-Aware Task Scheduling for MapReduce Applications in Virtual Clusters,” 22nd international symposium on High-performance parallel and distributed computing, pp. 227-238, 2013.

[33] Ripal Nathuji, Aman Kansal, Alireza Ghaffarkhah, “Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds,” 5th European conference on Computer systems, pp. 237-250, 2010.

[34] Lei Yang,Yu Dai, Zhang, “MapReduce Scheduler by Characterizing Performance Interference,” China Communications, Volume 13 , Issue 10 , pp. 253-262, 2016.

[35] Vasile, Mihaela-Andreea and Pop, Florin and Tutueanu, Radu-Ioan and Cristea, Valentin and Kolodziej, Joanna, “Resource-aware Hybrid Scheduling Algorithm in Heterogeneous Distributed Computing,” Future Generation Computer Systems, Volume 51, pp. 61-71, 2015.

[36] Ikken, Sonia and Renault, Eric and Kechadi, M. Tahar and Tari, Abdelkamel, “Toward scheduling I/O request of Mapreduce tasks based on Markov model,”

International Conference on Mobile, Secure and Programmable Networking, pp. 78-89, 2015.

[37] Q. Zhang and M.F.Zhani and Y.Yang and R.Boutaba and B.Wong, “PRISM: Fine-Grained Resource-Aware Scheduling for MapReduce,” IEEE Transactions on Cloud Computing, Vol. 3, pp. 182-194, 2015

[38] Yang, Shin-Jer and Chen, Yi-Ru, “Design Adaptive Task Allocation Scheduler to Improve MapReduce Performance in Heterogeneous Clouds,” Journal of Network and Computer Applications, Volume 57, pp. 61-70, 2015.

[39] Anjos, Julio and Izurieta, Ivan Carrera and Kolberg, Wagner and Tibola, Andre Luis and Arantes, Luciana and Geyer, Claudio, “MRA++: Scheduling and data placement on MapReduce for heterogeneous environments,” Future Generation Computer Systems, Vol. 42, pp. 22-35, 2015.

[40] Mao, Yingchi and Zhong, Haishi and Wang, Longbao, “A Fine-Grained and Dynamic MapReduce Task Scheduling Scheme for the Heterogeneous Cloud Environment,” 14th International Symposium on Distributed Computing and Applications for Business Engineering and Science, pp. 155-158, 2015.

[41] F.Yan and L.Cherkasova and Z.Zhang and E.Smirni, “DyScale: A MapReduce Job Scheduler for Heterogeneous Multicore Processors,” IEEE Transactions on Cloud Computing, Vol. 5, pp. 317-330, 2017.

[42] Lin, Wen-hui and LEI, Zhen-ming and Jun, YANG and Fang, LIU and Gang, HE and Qin, WANG, “MapReduce optimization algorithm based on machine learning in heterogeneous cloud environment,” The Journal of China Universities of Posts and Telecommunications, Volume 20, Issue 6, pp. 77-87, 2013.

[43] Verma, Abhishek and Cherkasova, Ludmila and Campbell, Roy H, “ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments,” 8th ACM International Conference on Autonomic Computing, pp. 235-244, 2011.

[44] Xie, Jiong and Yin, Shu and Ruan, Xiaojun and Ding, Zhiyang and Tian, Yun and Majors, James and Manzanares, Adam and Qin, Xiao, “Improving mapreduce

performance through data placement in heterogeneous hadoop clusters," Parallel and Distributed Processing, Workshops and Phd Forum, pp. 1-9, 2010.

[45] Zaharia, Matei and Borthakur, Dhruba and Sen Sarma, Joydeep and Elmeleegy, Khaled and Shenker, Scott and Stoica, Ion, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," 5th European conference on Computer systems, pp. 265-278, 2010.

[46] Tian, Chao and Zhou, Haojie and He, Yongqiang and Zha, Li, "A dynamic mapreduce scheduler for heterogeneous workloads," 8th International Conference on Grid and Cooperative Computing, pp. 218-244, 2009.

[47] Jianjiang Li, Yajun Liu, Jian Pan, Pend Zhang, Wei Chen, Lizhe Wang, "Map-Balance-Reduce: An improved parallel programming model for load balancing of MapReduce," Future Generation Computer Systems, 2017.

[48] Jaeseok Myung, Junho Shim, Jongheum Yeon, Sang-goo Lee, "Handling data skew in join algorithms using MapReduce," Expert Systems with Applications, Volume 51, pp. 286-299, 2016.

[49] Zhihong Liu, Qi Zhang, Reaz Ahmet, "Dynamic Resource Allocation for MapReduce with Partitioning Skew," IEEE Transactions on Computers, Volume 65, Issue 11, pp. 3304-3317, 2016.

[50] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, Alfons Kemper, "Handling Data Skew In Mapreduce," 1st International Conference on Cloud Computing and Services Science, pp. 574-583, 2011.

[51] F.A.N. Yuanquan, W.U.Weiguo, X.U.Yunlong, and C.Heng, "Improving MapReduce Performance by Balancing Skewed Loads," China Communications, Volume 11, Issue 8, pp. 85-108, 2014.

[52] W.Chen, I.Paik, and Z.Li, "Tology-Aware Optimal Data Placement Algorithm for Network Traffic Optimization," IEEE Transactions on Computers, Volume 65, Issue 8, pp. 2603-2617, 2016.

[53] V.Ubarhande, A.M.Popescu, “Novel Data-Distribution Technique for Hadoop in Heterogeneous Cloud Environments,”Ninth International Conference on Complex, Intelligent, and Software Intensive Systems, pp. 217-224, 2015.

[54] C.W.Lee, K.Y.Hsieh, S.Y.Hsieh, and H.C.Hsiao, “A Dynamic Data Placement Strategy for Hadoop in Heterogeneous Environments,” Big Data Research, Volume 1, pp. 14-22, 2014.

[55] Xueping Li, Kaike Zhang, “A hybrid differential evolution algorithm for multiple container loading problem with heterogeneous containers,” Computers and Industrial Engineering, Volume 90, pp. 305-313, 2015.

[56] Jesus Iglesias, Milan De Cauwerb, Deepak Mehtab, Barry O’Sullivan b, “Increasing task consolidation efficiency by using more accurate resource estimations,” Future Generation Computer Systems, vol. 56, pp. 407-420, 2016.

[57] Christian Bluma, Verena Schmidc, “Solving the 2D bin packing problem by means of a hybrid evolutionary algorithm,” International Conference on Computational Science, vol. 18, pp. 899-908, 2013.

[58] Georgios L.Stavrinides, Helen D.Karatza, “Scheduling real-time DAGs in heterogeneous clusters by combining imprecise computations and bin packing techniques for the exploitation of schedule holes,” Future Generation Computer Systems, vol. 28, pp. 977-988, 2012.

[59] Célia Paquay, Sabine Limbourg, Michaël Schyns, “A tailored two-phase constructive heuristic for the three-dimensional Multiple Bin Size Bin Packing Problem with transportation constraints," European Journal of Operational Research, vol. 267, pp. 52-64, 2018.

[60] Yusen Li, Xueyan Tang, Wentong Cai, "Dynamic Bin Packing for On-Demand Cloud Resource Allocation,” IEEE Transactions On Parallel And Distributed Systems, Vol. 27, No. 1, January 2016.

[61] Adam Stawowy, “Evolutionary based heuristic for bin packing problem,” Computers and Industrial Engineering, Vol. 55,pp. 465-474, 2008.

[62] Christine Bassema, Azer,“"Multi-Capacity Bin Packing with Dependent Items and its Application to the Packing of Brokered Workloads in Virtualized Environments,” Future Generation Computer Systems, vol. 72, pp. 129-144, 2017.

[63] Cong Liu, Sanjeev Baskiyar, “Scheduling Mixed Tasks with Deadlines in Grids using Bin packing,” 14th IEEE International Conference on Parallel and Distributed Systems, pp. 229-236, 2008.

[64] Lijun Wei, Wee-Chong Oon, Wenbin Zhu, Andrew Lim, “A goal-driven approach to the 2D bin packing and variable-sized bin packing problems,” European Journal of Operational Research, vol. 224, pp. 110-121, 2013.

[65] Alessio Trivella, David Pisinger, “The load-balanced multi-dimensional bin-packing problem,” Computers and Operations Research, vol. 74, pp. 152-164, 2016.

[66] D.S.Liu, K.C.Tan, S.Y.Huang, C.K.Goh, W.K.Ho, “On solving multiobjective bin packing problems using evolutionary particle swarm optimization,” European Journal of Operational Research, vol. 190, pp. 357-382, 2008.

[67] Marc P.Renaulta, AdiRoséna, Robvan Steeb, “Online algorithms with advice for bin packing and scheduling problems,” Theoretical Computer Science, vol. 600, pp. 155-170, 2015.

[68] Zhao-hong Jia, Joseph Y.T.Leung, “A meta-heuristic to minimize makespan for parallel batch machines with arbitrary job sizes,” European Journal of Operational Research, vol. 240, pp. 649-665, 2015.

[69] https://github.com/rathinaraja/Binpacking-using-ACO

[70] J.Liao, L.Zhang, T.Li, J.Wang, and Q.Qi, “Efficient and fair scheduler of multiple resources for MapReduce system,” IET Software, vol. 10, pp. 182-188, 2016.