Master's thesis

Improving the Resource Utilisation in MapReduce

（MapReduceにおけるリソース利用率の改善）

February 5th, 2015

Supervisor
Prof. Masaru Kitsuregawa

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo

48-136444 Kun Liu

Abstract

MapReduce has gained in widespread popularity over recent years. It achieves great scalability by subdividing jobs into tasks, distributing the tasks across the cluster and executing them in parallel. Since the advent of MapReduce, various effort has been made to improve the cluster resource utilisation. Current scheduling methods mostly focus on the resource sharing policy amongst jobs. However, resources in MapReduce are allocated to tasks rather than directly to jobs, yet work from the task point of view is relatively lacking.

In this thesis, I review a few recent advances that delve into the resource allocations on the task level. Despite their advantages, none of those approaches answer the questions raised by various workload patterns, such as CPU-intensive and I/O-intensive, in homogeneous environments. As such, I propose a Finer Grained CPU Scheduler that effectively improves the CPU utilisation, yet does not over-stress the CPU resources, by taking into account the diverse CPU requirements of tasks. Experiments conducted on a Hadoop cluster demonstrate that compared to state-of-art approach YARN, the Finer Grained Scheduler significantly improves the throughput of CPU-intensive workloads without compromising the performance of I/O-intensive ones.

Keyword: MapReduce, CPU, resource utilisation, scheduling

Acknowledgements

I would like to express my gratitude to my advisor Prof. Masaru Kitsuregawa, who has been a mentor and inspiration to me. I would like to thank Dr. Daisaku Yokoyama for his continuous support throughput not only the project but the entire two years of my Master's. Thanks to Dr. Miyuki Nakano for her valuable advice on this research. Thanks to Prof. Masashi Toyoda, Dr. Nobuhiro Kaji, Dr. Naoki Yoshinaga, Dr. Masahiko Itoh, and those who accompanied me in the presentation exercises, who made helpful comments during the weekly meetings, who joined me for regular discussions.

Thanks to Kitsuregawa Lab for providing a tremendous environment, with all the facilities indispensable for researches, and the academic atmosphere where I could be constantly motivated. Thanks to every member of Kitsuregawa Lab. They welcomed me with open arms when I first got in here, and treated me with nothing but kindness and consideration. Studying in a foreign country far away from family was not easy, but they made me feel at home.

A special thanks to my family, who have been extremely supportive even in rough times. Words cannot express how grateful I am to my parents, aunts and uncles. Thanks to all of my friends who supported me during this thesis.

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Resource Management in MapReduce . . . 3
  - 1.2.1 FIFO Scheduler . . . 5
  - 1.2.2 Size-based Scheduler . . . 7
  - 1.2.3 Capacity Scheduler . . . 7
  - 1.2.4 Fair Scheduler . . . 8
- 1.3 The Structure of This Thesis . . . 9

2 Related Work 10
- 2.1 YARN . . . 10
  - 2.1.1 Architecture . . . 11
  - 2.1.2 Resource Model . . . 13
- 2.2 ThroughputScheduler . . . 15
  - 2.2.1 Intuition . . . 15
  - 2.2.2 Task Model . . . 16
  - 2.2.3 Scheduling Policy . . . 18

3 Finer Grained Resource Allocation 19
- 3.1 CPU Under-utilisation in YARN . . . 20
- 3.2 Finer Grained CPU Resource Model . . . 24
  - 3.2.1 Parametrisation . . . 26
  - 3.2.2 CPU Requirement Analysis . . . 28
- 3.3 Implementation . . . 30
  - 3.3.1 System Design . . . 30

3.3.2 Scheduling Framework 32
3.3.2.1 Job Submission 32
3.3.2.2 Task execution 34
3.3.2.3 Job completion 36

4 Experiments **37**
4.1 Environment 38
4.2 Benchmarks 39
4.3 Metrics 41
4.4 Results 42
4.4.1 Results under Balanced Pattern 42
4.4.2 Results under CPU-heavy Pattern 43
4.4.3 Results under I/O-heavy Pattern 44

5 Conclusion **46**

Bibliography **48**

List of Figures

1.1 Simplified MapReduce framework . . . 3
1.2 Task slots in MapReduce . . . 4
1.3 Waiting time (s) of each job under FCFS algorithm . . . 6
1.4 Hierarchical queues in MapReduce . . . 8

2.1 JobTracker and TaskTrackers in classic MapReduce . . . 11

3.1 CPU allocations in YARN . . . 21
3.2 CPU usage of the datanode in Test 4 (where 11 tasks executed in parallel) . . . 22
3.3 Average MAP task time in each test (time in ms, lower is better; x axis shows the total number of parallel tasks in each test) . . . 23
3.4 CPU usage of the datanode in Test 0 . . . 24
3.5 CPU scheduling on a datanode: YARN vs. Finder Grained . . . 27
3.6 Simplified workflow of Finer Grained CPU scheduling . . . 33

4.1 Job execution times (in ms, lower is better) under balanced workload pattern . . . 43
4.2 Job execution times (in ms, lower is better) under CPU-heavy workload pattern . . . 44
4.3 Job execution times (in ms, lower is better) under I/O-heavy workload pattern . . . 45

List of Tables

1.1 Resource time (s) of three jobs 5

3.1 Datanode specifications 21

4.1 Experimental environment (per datanode) 37
4.2 dd test results (MB/sec) 38
4.3 Per MAP/REDUCE task CPU requirement of each workload 41
4.4 Average MAP task time (in ms, lower is better) under balanced workload pattern 42
4.5 Average MAP task time (in ms, lower is better) under CPU-heavy workload pattern 43
4.6 Average MAP task time (in ms, lower is better) under I/O-heavy workload pattern 44

Chapter 1

Introduction

1.1 Background

In this era of Big Data, distributed processing is becoming increasingly indispensable to processing a massive amount of data in a timely manner. It has motivated numerous tools such as MapReduce, which by virtue of scalability, fault-tolerance and simplicity, has become the de facto standard for large scale data analytics, and is gaining great momentum from both academia and industry.

The basic workflow of MapReduce is straightforward. From the users' perspective, operations are performed through a variety of jobs. Once a job gets submitted to the MapReduce framework, however, it is subdivided into MAP and REDUCE tasks. In particular, the input of the job is split evenly into blocks, and a MAP task is spawned for each block. When all MAP tasks finish, their outputs are partitioned, often by a hash function, into REDUCE tasks, which then process those outputs to yield the final results.

To MapReduce, a framework designed for executing multiple jobs and tasks in parallel, resource sharing is of crucial importance. In MapReduce, multiple jobs often need to compete for the cluster resources, and resources assigned to each job are shared by its tasks. By distributing the tasks to a large number of datanodes across the cluster and executing them in parallel, the execution time of a job could be significantly reduced. This mechanism enables great scalability and lies at the

heart of MapReduce. Back in 2008, Yahoo! reported that their Hadoop[1] cluster could scale out to 4,000 nodes [1]. In the same year, Hadoop sorted 1 terabyte of data in 209 seconds, beating the previous record of 297 seconds in the annual general purpose (Daytona) terabyte sort benchmark. It was the first time ever that either a Java or an open source program had won [2].

The idea of resource sharing is not new. Long before the advent of MapReduce, it had enabled multitasking in most modern operating systems. By sharing common computing resources in the system, such as CPUs and disks, multiple tasks are able to progress simultaneously. Take CPU resources for instance. On the very bottom level, a CPU is only capable of carrying out one operation at a particular time, but it constantly makes clever decisions to switch back and forth amongst the runnable tasks. Each task thus gets a proportion of the CPU time. Since all these are done under the hood, from the perspective of the user, it appears that multiple tasks are running concurrently.

Compared to the simple example above, resource sharing in MapReduce is much more complicated, with a few extra factors having to be considered

- **Multiresource** It is relatively straightforward to manage just one single resource such as CPU. But in real world, most of the time several different kinds of resources are shared at the same time. In addition, a task generally requires more than one kind of resources, e.g., nearly all tasks require CPU for computations, some need disks for reading or/and writing files, and some even need network for transferring data remotely

- **Diverse requirements** Tasks achieve various objectives, therefore they have diverse resource requirements. Based on the requirements, tasks could be classified into CPU-bound, I/O-bound, memory-bound, etc. For a CPU-bound task, its execution time is primarily determined by the speed of the processor, which also implies that upgrading the CPU could possibly improve the task performance. The same idea applies to other kinds of resources. In this thesis, I only consider CPU-bound and I/O-bound tasks

- **Distributed resources** In a distributed environment like MapReduce, resources are scattered across the cluster, adding to the complexity of resource

[1] Apache Hadoop is the open-source software framework that mainly consists of Hadoop Distributed File System (HDFS) and MapReduce.

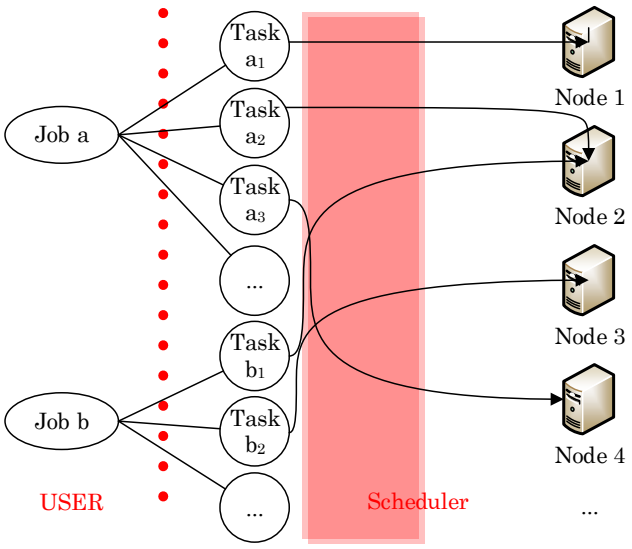


FIGURE 1.1: Simplified MapReduce framework

management. Moreover, compared to non-distributed environments such as a single machine, a cluster usually has an enormous amount of resources to handle, which consequently raises the scalability concern, with the resource management module being the bottleneck

1.2 Resource Management in MapReduce

To manage resources amongst multiple concurrent jobs and tasks, decisions have to be constantly made on resource allocations. Cluster resources are limited compared to the endless needs, therefore such decisions could have a huge impact on the resource utilisation and the overall throughput of the cluster.

Resource management is mostly automatic and transparent, e.g., in the case of an operating system, it usually rests with the kernel to make decisions on resource allocations. Similarly in MapReduce, the *scheduler* module is in charge of resource management. As illustrated in FIGURE 1.1, the user defines[2] and submits his jobs, and leaves the rest - subdividing each job into tasks, distributing the tasks across the cluster, and executing them - to the MapReduce framework. This characteristics, often referred to as "simplicity", is one of the main advantages of MapReduce. Preserving such simplicity is a vital principle in my work.

[2]A MapReduce job is usually defined by writing the *Mapper* and *Reducer* classes, which specify the behaviour of the MAP and REDUCE tasks.

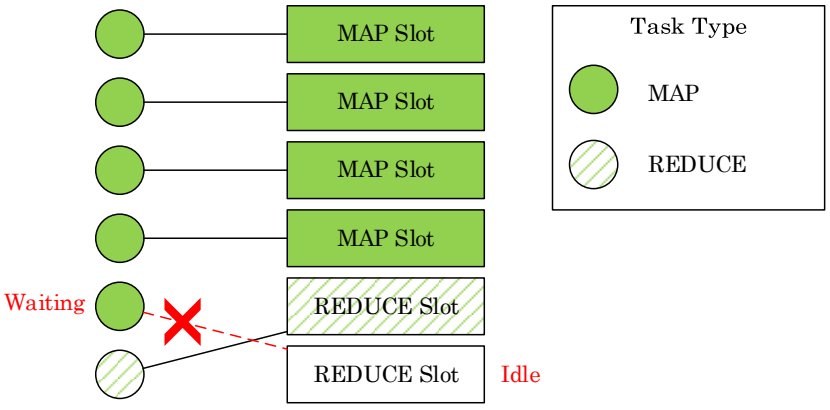


FIGURE 1.2: Task slots in MapReduce

In classic MapReduce, cluster resources are allocated as "task slots" [3, 4]. Each datanode is configured with a fixed number of MAP slots and REDUCE slots, each of which could be allocated to a MAP / REDUCE task. There are two key shortcomings in this static slot approach

- **Lack of flexibility** In MapReduce, a MAP task cannot be executed in a REDUCE slot, and vice versa. As a result, cluster resources could be wasted in scenarios like FIGURE 1.2, where some REDUCE slots are left idle while there are MAP tasks waiting to be scheduled, or the opposite
- **Inefficient memory utilisation** Memory is the inelastic resource in MapReduce, i.e., without sufficient memory a task simply cannot be successfully executed. Consequently, under the static slot strategy, the number of slots per datanode is limited by the most memory-demanding tasks

Consider the following example of memory allocation in classic MapReduce. Assume that

1. Each datanode has 8 GB memory in total for executing MAP tasks
2. Job A requires 1 GB memory per MAP task
3. Job B requires 2 GB memory per MAP task

To provide the guarantee that tasks would never fail due to insufficient memory, the number of MAP slots per datanode, i.e., the maximal number of parallel MAP tasks per node is

$$\#\ MAP\ slots\ per\ node = \frac{8GB}{2GB} = 4$$

Job	Resource Time
J1	30
J2	5
J3	5

TABLE 1.1: Resource time (s) of three jobs

i.e., every MAP task is presumed to require 2 GB memory. This assumption leads to inefficient memory utilisation, but had the number of MAP slots per datanode been set to a higher value, tasks could potentially fail due to insufficient memory. e.g., assume each datanode is configured with 5 MAP slots. If a particular datanode had been allocated 5 MAP tasks, 4 of which were from job B (and the 5th might come from job A or job B), then the actual memory requirements would have been

$$Requirement \geq 1GB + 2GB \times 4 = 9GB > 8GB = Capacity$$

which would always exceed the memory capacity of the datanode, consequently causing tasks to fail

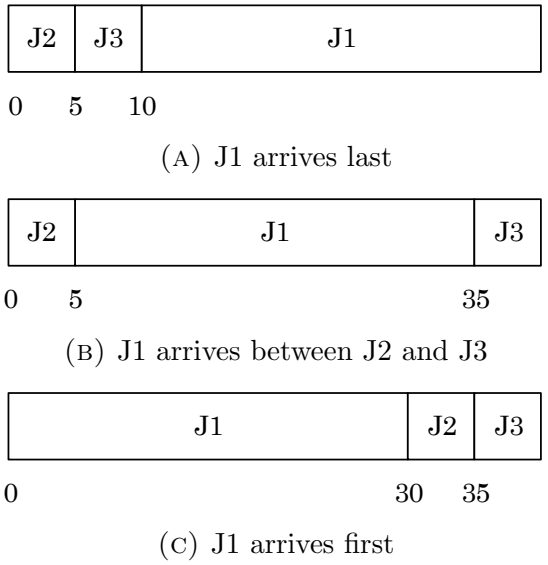
Since multiple jobs often have to compete for the cluster resources, there needs to be a policy to address the resource sharing amongst jobs. In particular, decisions have to be constantly made on the selection of job to assign resources to next. In the following subsections, I review some scheduling approaches that frequently appear in MapReduce.

1.2.1 FIFO Scheduler

FIFO Scheduler in MapReduce closely resembles the FCFS (First-Come-First-Served) policy that appears in many scheduling problems. Under the FIFO policy, jobs are scheduled based strictly on the order of submission.

FIFO is straightforward, but it has the shortcoming of yielding long average response time[3] under certain circumstances. Consider an example of three jobs, as shown in TABLE 1.1. For a particular job, its resource time refers to the amount of time it needs to use the whole cluster to complete. Although resources in MapReduce are scheduled to tasks rather than directly to jobs, here I make such

[3]The response time for a job refers to the difference between its submission time and completion time.



(A) J1 arrives last

(B) J1 arrives between J2 and J3

(C) J1 arrives first

FIGURE 1.3: Waiting time (s) of each job under FCFS algorithm

simplifications to better illustrate the idea. Assume that the three jobs arrive at the same time in a particular order

- If J1 arrives last, as illustrated in Gantt chart 1.3a, the average waiting time is $(0 + 5 + 10)/3 = 5s$
- Had J1 arrived between J2 and J3, as illustrated in FIGURE 1.3b, the average waiting time would have been $(0 + 5 + 35)/3 = 13.3s$
- The worst case is when J1 arrives first, as shown in FIGURE 1.3c, yielding an average waiting time of $(0 + 30 + 35)/3 = 21.6s$

Having a short job like J2 or J3 wait for a duration much longer than the job itself is by no means reasonable.

In an operating system, the FIFO (or FCFS) policy could cause the nasty "convoy effect" [5], i.e., a long process holds the resources, blocking all the other processes. It leads to poor utilisation of the other kinds of resources in the system, e.g., many I/O bound processes could get stuck behind one single CPU-bound process, consequently leaving the I/O resources idle. Similar scenarios could occur in MapReduce, where a CPU-intensive job hogs the cluster, preventing the I/O resources from being efficiently utilised.

1.2.2 Size-based Scheduler

Proposed by Pastorelli Mario, et al. [6], the Size-based Scheduler is basically an SRTF (Shortest-Remaining-Time-First) emulation on MapReduce. For a particular job, its size is the aggregation of estimated execution time of all its tasks. Jobs with smaller *remaining sizes* are prioritised over those with bigger remaining sizes. In particular, if a new job J_{new} arrives with resource time shorter than the remaining time of currently executing job $J_{current}$, $J_{current}$ is preempted[4].

SJF is theoretically one of the most efficient scheduling algorithms in terms of average response time [7]. This is good for the resource utilisation - more jobs could complete within the same time. However, it faces a critical challenge in reality - the resource time of a newly arrived job is generally unknown. To tackle this problem, Size-based Scheduler estimates the size of a new job based on its first few tasks.

Furthermore, to prevent starvation of long jobs, SJF adopts the job *aging* approach, i.e., the remaining size of a waiting job virtually[5] reduces - as if it were actually progressing - the longer it waits, eventually small enough for the job to be scheduled.

1.2.3 Capacity Scheduler

Developed by Yahoo!, the Capacity Scheduler [8] provides a solution for sharing a large cluster securely amongst multiple tenants, by introducing the concept of *queues*. Specifically, it forms hierarchical queues, as illustrated in FIGURE 1.4: all queues descend from the root queue, which is granted the resources of the entire cluster; for each queue, including the root, its resources are distributed amongst its child queues recursively; jobs only run in the leaf queues, which cannot have any child queues. Within each leaf queue, jobs are usually scheduled under the FIFO policy.

In real world, typically each organisation has its private compute resources that is able to satisfy its peak-time requirement. But this leads to poor utilisation on

[4]Strictly speaking, Size-based Scheduler does not allow pre-emption of tasks. The pre-emption here simply means that 'the next resource will be allocated to J_{new} rather than $J_{current}$'.

[5]In Size-based Scheduler, this is referred to as the *virtual remaining time* of a job.

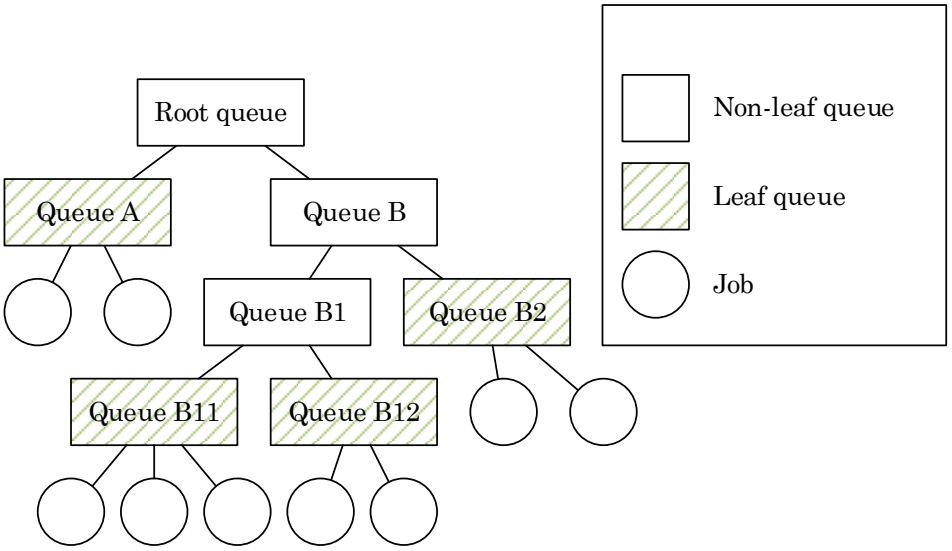


FIGURE 1.4: Hierarchical queues in MapReduce

average. By sharing resources on a large cluster amongst organisations, the overall utilisation could be largely improved.

1.2.4 Fair Scheduler

Developed by Facebook, Inc., Fair Scheduler [9] assigns resources to jobs such that all jobs get, on average, an equal share of resources over time. It enables short jobs to finish in reasonable time while not starving long ones. This is beneficial to the utilisation of the cluster resources since it allows a better chance of running mixed workloads, i.e., CPU-intensive jobs as well as I/O-intensive ones, therefore leading to a more balanced utilisation of multiple resources.

If configured with the size based weight policy, Fair Scheduler is loosely akin to the priority-based scheduling algorithm [10]. In fact, Size-based Scheduler could also be viewed as a special case of priority-based algorithm, i.e., the smaller a job is, the higher its priority.

Like Capacity Scheduler, Fair Scheduler supports hierarchical queues. In addition, to provide increased flexibility, it allows every queue to customise its own resource sharing policy, such as FifoPolicy, FairSharePolicy, etc.

1.3 The Structure of This Thesis

Section 1.2 reviews some commonly adopted scheduling approaches. When it comes to selecting the "best" scheduling method under particular circumstances, various criteria need to be considered, therefore it is difficult to assert that one scheduler is superior to another. Even the "problematic" FIFO is useful in some situations, e.g., within a queue in the Fair Scheduler. However, as a rule of thumb and a primary objective, a scheduler should always make good use of the cluster resources. Improving the resource utilisation in MapReduce is thus the focal point throughout this thesis.

Current scheduling algorithms mostly focus on the job level. They specify the resource sharing policy amongst multiple jobs, or more specifically, the decision making strategy on the selection of job to schedule resources to next. However, resources in MapReduce are allocated to tasks rather than directly to jobs, yet relatively speaking, researches on improving the resource utilisation from the task point of view are lacking. In this thesis, I address the resource utilisation problem by delving into the resource allocations to tasks, and improve the utilisation by taking into account the diverse task requirements.

The rest of this thesis is organised as follows

- **Chapter 2** introduces a few related researches on improving the resource utilisation in MapReduce. They differ from the scheduling algorithms in this chapter in that they focus on the resource allocations to tasks, rather than resource sharing policy amongst jobs
- **Chapter 3** describes the approach of my work - Finer Grained CPU Scheduling. I firstly discuss the problem of CPU under-utilisation in MapReduce, then propose a refined CPU resource model, and finally design and implement a scheduling system that puts the model into practical use
- **Chapter 4** summarises the experiments for evaluating the Finer Grained CPU Scheduling
- **Chapter 5** concludes this thesis, and discusses future work, including a few possible extensions on the Finer Grained CPU Scheduling

Chapter 2

Related Work

In this chapter I present related work that aims to improve the cluster resource utilisation in MapReduce. Firstly, I walk through YARN (Yet Another Resource Negotiator [11], developed by Yahoo!), which revolutionised the mechanism of resource allocations. Next, I introduce the ThroughputScheduler propose by Shekhar Gupta et al. [12]. It learns the resource requirement profile of jobs, and schedules resources accordingly. There are other researches that adopt a similar learning-based approach, such as CASH (Context Aware Scheduler for Hadoop [13]), but in this chapter I only cover the ThroughputScheduler as the representative work.

2.1 YARN

Section 1.2 describes the "static task slot" strategy for resource allocations in classic MapReduce, which is rather inflexible and inefficient. The fundamental problem is that from the perspective of the scheduler, every MAP task is treated equally and so is every REDUCE task, regardless of their diverse resource requirements. In addition, the strict task-slot mapping, i.e., the restriction that a MAP task cannot be executed in a REDUCE slot and vice versa, leads to resource waste in scenarios like 1.2, where REDUCE slots are left idle while there are waiting MAP tasks, or the opposite.

To overcome such shortcomings, Yahoo! designed and developed the next generation of Hadoop's compute platform, YARN. In the following subsections, I firstly

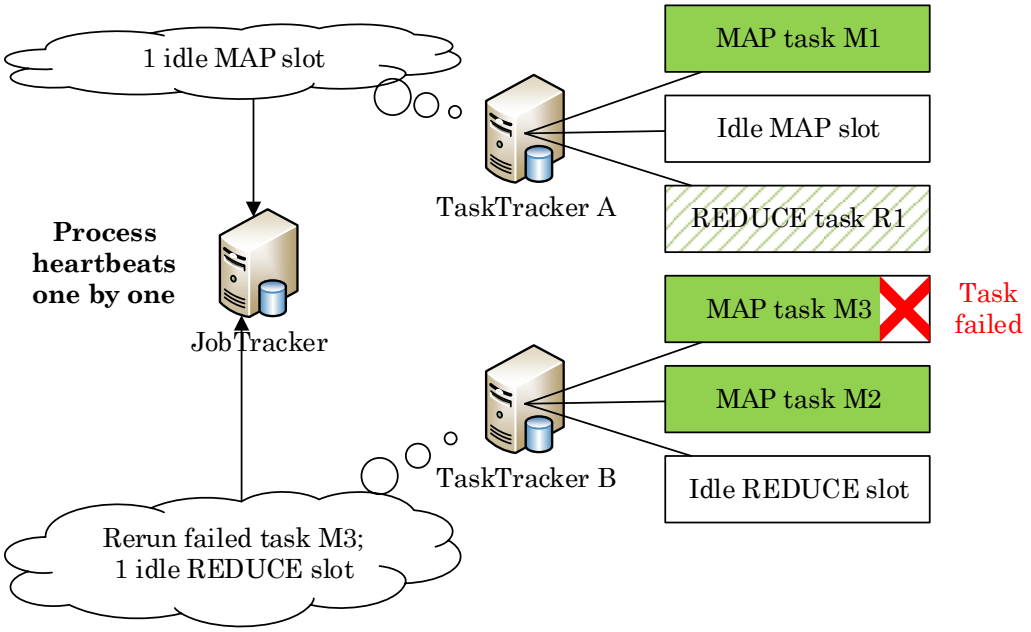


FIGURE 2.1: JobTracker and TaskTrackers in classic MapReduce

compare the basic architecture of YARN with classic MapReduce, then describe in details the resource management in YARN, and why it is more efficient.

2.1.1 Architecture

In classic MapReduce, all the scheduling decisions are made by the central *JobTracker*, which usually runs along with the namenode[1] on the dedicated master node. Each slave node (datanode) runs a *TaskTracker* which manages the task slots configured with it. Specifically, it starts JVM processes for tasks to run within, monitors their status and progress, and provides crucial functionalities such as fault-tolerance, i.e., when a task fails, the relevant TaskTracker would request the JobTracker for re-execution. It also notifies the JobTracker, typically every few minutes, to reassure the JobTracker about its liveness. A TaskTracker communicates with the JobTracker by sending periodic heartbeats, as illustrated in FIGURE 2.1. Upon receiving a heartbeat, the JobTracker locks the scheduler until the heartbeat is processed, i.e., it could process only one heartbeat at a particular time.

[1] As opposed to the datanodes which store the actual data, the namenode is the coordinator of the HDFS file system. It keeps the directory tree of all files, and tracks how these files are divided into blocks and distributed across the cluster.

The classic MapReduce framework has the following problems

- **Scalability** Since the JobTracker listens to all TaskTrackers in the cluster, this architecture raises the scalability concern - the workload of the JobTracker is roughly proportional to the total number of TaskTrackers, consequently as the cluster grows larger and larger, the JobTracker could readily be overwhelmed by the sheer amount of information it needs to process
- **Reliability** In classic MapReduce, the JobTracker is the single point of failure, i.e., if the JobTracker goes down, it would take the whole cluster with it - all the running jobs would be lost, and it would take a herculean effort to manually recover them. Furthermore, the fact that the JobTracker handles the resource allocations all by itself and processes heartbeat updates from all datanodes makes it even more vulnerable

To tackle these two issues, YARN decouples a part of the scheduling function from the JobTracker. The new centrepiece is now called the *Resource Manager* (RM). The name is self-explanatory: it is dedicated to resource management. In particular, RM monitors the resource usage of the datanodes across the cluster; it also receives resource requests from jobs, and informs them which datanode has the requested resources.

The primary difference between YARN and classic MapReduce is the per-job *Application Master* (AM), which coordinates the execution of the job in the cluster. When a job gets started, the RM launches an AM for it, and leaves the job semantics to the AM thereafter. In particular, the AM sends periodic heartbeats to the RM to affirm the liveness of the job and requests resources for its task. Once the AM receives a response from the RM that informs it which datanode has the requested resources, it launches a task in the right place. In addition, the AM monitors the status and progress of the tasks, and provides fault-tolerance, speculative execution[2], etc. In classic MapReduce, all these are handled by the single point of failure JobTracker. By delegating these responsibilities to the per-job AM, the burden on the new centrepiece RM is tremendously reduced.

[2]For a particular job, as most of its tasks are coming to a close, the scheduler would speculatively re-executes the *stragglers*, i.e., tasks that are progressing more slowly than others, in the hope that the speculative tasks would beat the original ones which are likely suffering poor performance, therefore reducing the response time of the job [14].

This re-designed architecture achieves much greater scalability than classic MapReduce, which was indeed one of the initial goals of YARN. Yahoo! claims that their cluster could be extended by YARN to over 7,000 nodes, which is an astounding advancement compared to 4,000 nodes, their largest cluster size before YARN.

More importantly, due to the largely reduced burden on the central RM, the cluster is capable of handling a much bigger number of jobs and tasks at the same time. Yahoo! reported the statistics on a 2,500 node cluster, in which the daily number of jobs went from about 77k under classic MapReduce, to roughly 100k on YARN. Similarly the daily number of tasks went from 4M to approximately 10M. This implies that YARN significantly improves the resource utilisation of the cluster and therefore yields much higher throughput.

2.1.2 Resource Model

Apart from the refined MapReduce framework, another valuable contribution of YARN is its new resource model. As described in Section 1.2, classic MapReduce completely ignores the diverse resource requirements of tasks, and allocates cluster resources as static task slots, which is not only inflexible but inefficient. YARN tackles such problems by introducing the concept of *container*.

In YARN, a container is a logical bundle of resources that could be allocated for a task. Currently it models both memory and CPU. This resource model differentiates from classic MapReduce mainly in two aspects, each of which corresponds to one issue described in Section 1.2

- **Increased Flexibility** In YARN, there are no such concepts as "MAP container" or "REDUCE container". As long as the resources wrapped in the container meet the requirements of the task, or more specifically, the amount of memory and the number of CPU cores that the task requests, an allocation could be arranged. As a result, the nasty resource waste problem illustrated in FIGURE 1.2 could be completely avoided
- **Improved Efficiency** Before each job is submitted to the centrepiece RM, it could configure its resource requirement per MAP task and per REDUCE task. If no such configuration are made, default values[3] are used instead. Unlike classic MapReduce, this mechanism takes into consideration the diverse

[3]The default values for resource requirement per MAP/REDUCE task are also configurable.

resource requirement of tasks, and allows a dynamic per-job configuration, which effectively improves resource utilisation

Memory is the inelastic resource in the cluster. Consequently, in practice the memory request for a task normally needs to be set slightly higher than its actual requirement, in order to guarantee that the task would not fail due to insufficient memory. Furthermore, a task might "under-report" its memory requirement, accidentally or intentionally. As a result, its memory usage exceeds the amount planed for it by the RM. At best, the task itself fails; a worse case is that it drags other tasks down with it. To address this problem, YARN comes with a memory monitoring feature, which keeps track of the memory usage of each task, and enforces the killing of the task if its memory usage exceeds the request.

To illustrate the advantages of the resource model in YARN, consider such an example in which

1. Each datanode has 16 GB memory in total for executing tasks
2. Job A requires 1 GB per MAP task, 2 GB per REDUCE task
3. Job B requires 2 GB per MAP task, 1 GB per REDUCE task

With the static task slot approach of MapReduce 1, an appropriate scheme is to split the memory of each datanode evenly into two halves, one half for MAP tasks and the other for REDUCE tasks. To guarantee that tasks would never fail due to insufficient memory, the scheduler would have to presume that every task has the same requirement as the most memory-demanding tasks. Therefore, the maximal number of MAP/REDUCE task slots per datanode is

$$\begin{aligned} \#\ MAP\ slots &= \frac{16GB/2}{2GB} = 4 \\ \#\ REDUCE\ slots &= \frac{16GB/2}{2GB} = 4 \end{aligned}$$

However, had any datanode been assigned a MAP task of job A, the memory of that node would be wasted, as the actual memory usage of MAP tasks

$$MAP\ task\ memory\ usage \leq 1GB + 2GB \times 3 = 7GB < 8GB$$

is always smaller than the capacity. The same happens when the datanode is assigned a REDUCE task of job B. In addition, as previously described, distinguishing between MAP and REDUCE slots could also cause nasty resource waste problems.

With the new resource model of YARN, these kinds of waste could be effectively alleviated. As long as a datanode has memory left, it is considered by the RM as an option for task scheduling. If its remaining memory could satisfy a particular task, an allocation could be arranged.

2.2 ThroughputScheduler

As described in Section 2.1, YARN significantly improves the resource utilisation in MapReduce with its refined scheduling architecture and resource model. However, while it does consider the resource requirement of tasks, such consideration is largely limited to the inelastic memory resources. It does not answer the questions raised by various job patterns of MapReduce.

In a typical MapReduce environment, normally a variety of jobs are mixed together. Some jobs are CPU-intensive, i.e., they spend a large proportion of the time doing computations, and often need large amounts of memory to hold intermediate results; others are I/O-intensive, i.e., they do not require much processing power, but heavily rely on the disk performance of the datanodes. There are other categories, but for this thesis I only consider CPU- and I/O- intensive jobs.

In the next few subsections I introduce the ThroughputScheduler, which improves the cluster resource utilisation by actively matching job requirements to node capabilities, therefore reducing the overall job completion time. It is mainly targeted at clusters with heterogeneous nodes, which is not the case in my environment. Nevertheless, the idea of learning and utilising the resource requirement profile of jobs is generic and inspirational.

2.2.1 Intuition

In recent years, heterogeneous environments are becoming increasingly common. A typical example is virtualised data centres, such as Amazon's Elastic Compute

Cloud (EC2), according to [14]. In such environments, nodes have different hardware capabilities, e.g., some nodes may have relatively fast CPU but slow disks, while others are the opposite. Assigning a CPU-intensive task to the latter is by no means reasonable. The task performance would be degraded by the slow CPU, and the fast disks would be left idle, which is a tremendous waste of valuable resources. Unfortunately, MapReduce itself is not capable of distinguishing between CPU-intensive and I/O-intensive jobs, nor does it have a mechanism to take into account the node heterogeneity. ThroughputScheduler addresses such problems in the following steps

- **Node Capabilities** To get the knowledge of the node capabilities, a few probe jobs are executed. This is done offline, since hardware change is usually infrequent. Then with the capabilities, the resource requirements of jobs could be parametrised as well
- **Job Requirements** A MapReduce job is subdivided into many MAP and REDUCE tasks[4]. For a particular job, its MAP tasks are uniquely defined by a Mapper class, i.e., all its MAP tasks are in essence the same set of operations performed on different inputs. In addition, for the same job, the MAP tasks have nearly the same input data size, as each MAP task usually processes one HDFS block, which is of the fixed size for a particular job. Therefore, it is appropriate to assume that the MAP tasks that belong to the same job are very similar. It is thus reasonable to estimate the resource requirement of a job by analysing a small number of sample MAP tasks
- **Active Matching** With information obtained in the first two steps, ThroughputScheduler schedules tasks by optimally matching job requirements to node capabilities. In particular, it assigns CPU-intensive tasks to nodes with "relatively faster" CPU, and I/O-intensive tasks to nodes with "relatively faster" disks

2.2.2 Task Model

To enable the ThroughputScheduler, node capabilities (CPU and disk) and job requirements need to be translated into resource parameters. However, MapReduce itself does not provide such mechanisms. The task model, which does such

[4]It is possible for a job to have no REDUCE tasks.

parametrisations, lies at the heart of ThroughputScheduler. The actual calculations in the model are highly complicated. For understandability, here I only summarise the basics.

For a particular node n, the following capabilities are analysed

- Ω^n: the overhead (time in seconds) to start a task on n
- κ_c^n: the CPU capability of n
- κ_d^n: the disk capability of n

The value of Ω^n is estimated by executing a 'unit' MAP task that has an empty **map()** function. It has zero CPU requirement and zero disk requirement, thus its execution time could be an indicator of Ω^n.

The values of κ_c^n and κ_d^n are calculated in such fashion

1. Run a base job on each node. Assume its CPU and disk requirement per MAP task to be θ_c and θ_d respectively, then its average MAP task time on a particular node i is
$$T_1^i = \frac{\theta_c}{\kappa_c^i} + \frac{\theta_d}{\kappa_d^i} + \Omega^i \tag{2.2}$$
2. Run another job on each node. This job does the exact same I/O operations as the base job, but repeats the CPU computations for r times, therefore its CPU and disk requirement per MAP task is $r\theta_c$ and θ_d respectively. The average MAP task time of the new job on a particular node i is
$$T_r^i = \frac{r\theta_c}{\kappa_c^i} + \frac{\theta_d}{\kappa_d^i} + \Omega^i \tag{2.3}$$
3. Solving Equation 2.2 and 2.3 yields
$$\kappa_c^i = \frac{\theta_c(r-1)}{T_r^i - T_1^i} \tag{2.4}$$
where θ_c is still unknown
4. Choose an arbitrary node as the reference node. Assume $\kappa_c^1 = 1$, then θ_c becomes known, therefore θ_c^i for any i could be calculated by Equation 2.4
5. Calculate κ_d in the same fashion

Once the node capabilities become known, the resource profile of each job could be estimated by a small number of sample MAP tasks, and expressed as multiples of κ_c^1 and κ_d^1, i.e., the capability of the reference node.

2.2.3 Scheduling Policy

ThroughputScheduler schedules tasks based on firstly the data locality, then the optimal matching of job requirements and node capabilities. The data locality refers to the vital principle that MapReduce tries to assign tasks to nodes where the data to be processed is stored [15]. For each node n in the cluster, ThroughputScheduler does the following

1. If any data-local tasks could be scheduled, select a task from a job that optimally matches the capabilities of node n
2. If any rack-local[5] tasks could be scheduled, select a task from a job that optimally matches the capabilities of node n
3. Schedule a remote task based on the optimal matching

The optimal matching is to assign CPU-intensive tasks to nodes with "relatively faster" CPU, and I/O-intensive tasks to nodes with "relatively faster" disks. e.g., assume node n_1 has $\kappa_c^{n_1} = 2$ and $\kappa_d^{n_1} = 1$; node n_2 has $\kappa_c^{n_2} = 3$ and $\kappa_c^{n_2} = 3$; then n_1 has relatively faster CPU compared to its disk, despite the fact that n_2 has better CPU performance.

[5]Rack-local means that the input data is not on the node where the task is executed, but on some node in the same rack.

Chapter 3

Finer Grained Resource Allocation

Chapter 1

As described in Chapter 2, YARN revolutionises the mechanism of resource management in the cluster. It delegates a large part of the scheduling function to the per-job AM, thus allowing the centrepiece RM to dedicate to the resource management, and refines the resource model to enable flexible and more efficient allocations. As a result, resource utilisation is significantly improved in YARN. However, YARN does not take into account the actual CPU or disk resource requirements of various workloads. At present, disk resources are not parametrised. In addition, although YARN does model CPU resources, current CPU allocation is very coarse.

ThroughputScheduler does consider the resource requirements of jobs, but it is strictly limited the heterogeneous environments. It improves the resource utilisation by assigning resources to tasks that need it most, i.e., faster CPUs to CPU-intensive tasks and faster disks to I/O-intensive tasks. Unfortunately, this logic does not apply to a homogeneous cluster.

In this chapter, I present my work, the Finer Grained Scheduler. It models the resources in a homogeneous cluster in such a way that they reflect the node capabilities as well as the task requirements. In addition, to make use of the new resource model, I design a practical scheduler, and implement its features into Hadoop framework.

I found disk resources difficult to parametrise at the moment. In particular, even with only one I/O-intensive task, the disk bandwidth on the relevant datanode[1] would be almost saturated. In practice, however, it would be too harsh to allow only one or two I/O-intensive tasks on each datanode. Duggal et al. [16] reported that in some environments, it had been set up to 300 tasks per node for very CPU-light MAP tasks. Due to such reasons, I decided to focus on the CPU resources in my work, and consider I/O as a future extension.

3.1 CPU Under-utilisation in YARN

As described in Section 2.1.2, resources in YARN are allocated as containers. A container is a logical bundle of cluster resources, or more specifically, a certain amount of memory and an integer number of CPU cores. Each job is able to dynamically configure its resource request per MAP task and per REDUCE task before submitted to the RM. Memory resources are measured in megabytes, therefore YARN allows a relatively flexible and fine grained memory allocation. However, it is not the case for the CPU resources.

In YARN, CPU resources are allocated in the form of *virtual CPU cores* (vcore). For each datanode, the vcore capacity is configured equal to its number of physical cores. Each task requests exactly one vcore, i.e., every task is assumed to saturate one physical core. This strategy is loosely akin to the static task slots in classic MapReduce, which assumes that every task is equal, or more specifically, every task requires the same resources as the most resource-demanding tasks.

If workloads are all compute intensive, i.e., every task nearly saturates one physical core, the CPU scheduling of YARN works fine. In practice, however, there are both CPU- and I/O intensive workloads [17]. For an I/O intensive task, assuming it also saturates one physical core is not reasonable, as in fact the assigned CPU core is idle for most of the task's duration waiting for I/O operations. As illustrated in FIGURE 3.1, for each physical core assigned to an I/O intensive task, a large proportion of its processing power is left unutilised, which is a huge waste of valuable CPU resources.

[1]The relevant datanode is not necessarily the one where the task is executed - in distributed file systems such as HDFS, the data might be read from or written to another datanode in the cluster.

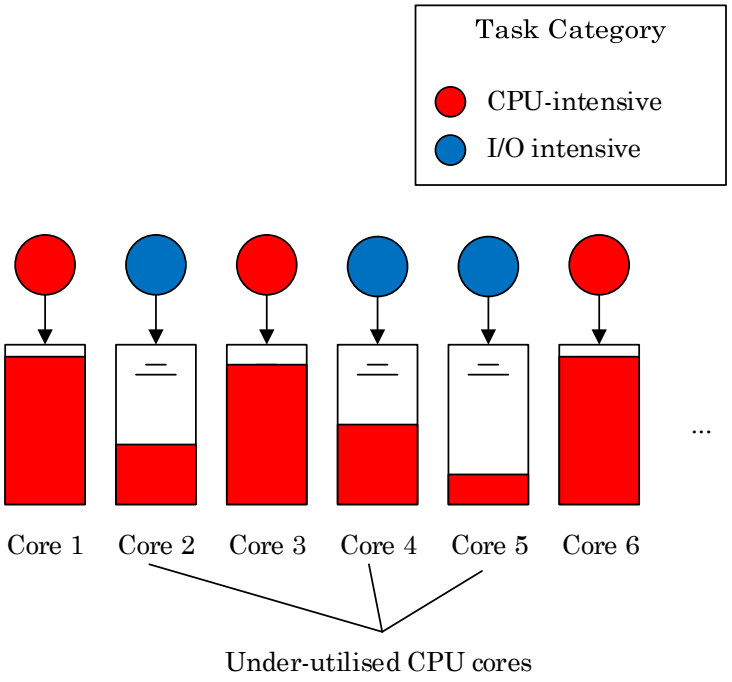


FIGURE 3.1: CPU allocations in YARN

Hardware	
CPU	Xeon E5530, 2.40 GHz, 8 cores
Memory	24 GB
Disk	500 GB
Software	
Hadoop	2.6.0
Framework	YARN
Java	Oracle 1.7.0_21

TABLE 3.1: Datanode specifications

I verify the problem of CPU under-utilisation in practice by performing a series of tests on a Hadoop cluster with a single datanode. The hardware specifications of the datanode are summarised in TABLE 3.1.

To resemble the real-world balanced workload pattern, I chose *Pi* and *TeraGen* as the CPU- and I/O intensive workload respectively. Both workloads came from the official Hadoop benchmark[2]. For each of the following tests, I would repeat for 10 times to calculate the average, median, etc.

Firstly in Test 1, I strictly followed the CPU scheduling policy of YARN and executed 2 jobs in parallel: a *TeraGen* job with 3 MAP tasks, each of which

[2]These workloads could be found in the hadoop-mapreduce-examples-⟨version⟩.jar shipped with Hadoop. They are also available in [18]

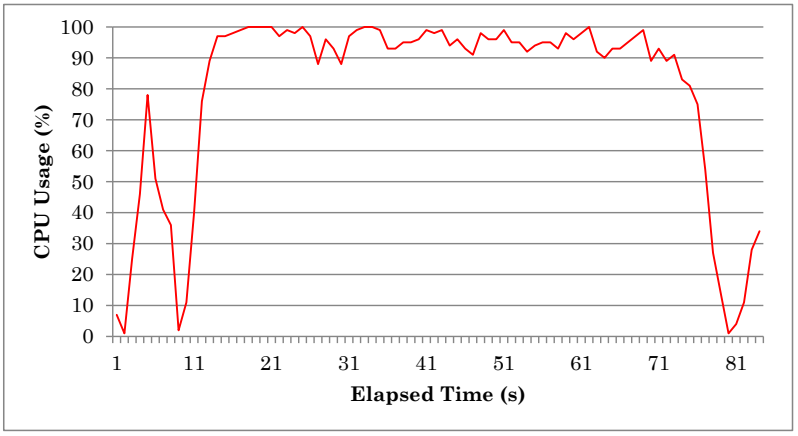


FIGURE 3.2: CPU usage of the datanode in Test 4 (where 11 tasks executed in parallel)

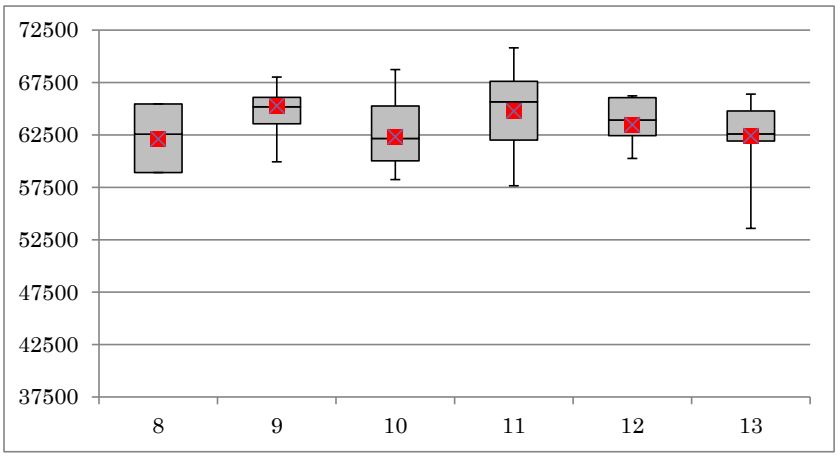
generates 1 GB of data; a *Pi* job with 3 MAP tasks, each of which generates 1.5 billion points. Since the datanode has 8 physical CPU cores in total, the scheduler would assume that the CPU resources were fully occupied by the 2 jobs (or more specifically, 8 cores were saturated by 6 MAP tasks and 2 per-job AMs).

From Test 2, I set the vcore capacity of the datanode to a very high value, so that current CPU policy would take no effect, thus more than 8 tasks were able to run in parallel. In addition, I gradually tuned up the number of MAP tasks in the *Pi* job. In particular, in Test 2 *Pi* contained 4 MAP tasks, in Test 3 *Pi* contained 5 MAP tasks, etc. The *TeraGen* job remained unchanged in every single test.

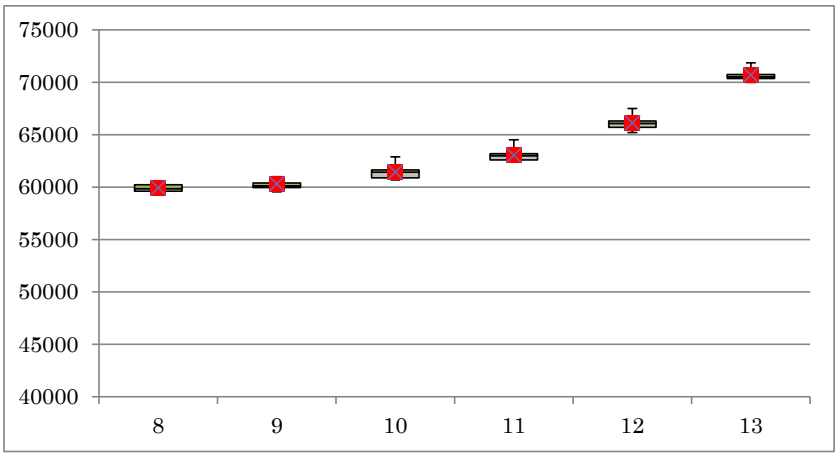
I monitored the CPU load of the datanode in each test. As illustrated in FIGURE 3.2, in Test 4 where 11 tasks were executed in parallel, the CPU load occasionally hit 100%, but was not overstressed. Thus, I conjectured that the task performance should not suffer significantly in Test 4.

As shown in box-and-whisker[3] FIGURE 3.3a, there was no performance degradation in the MAP tasks of *TeraGen*. When running with 8 Pi tasks (where x = 13 in FIGURE 3.3a), the MAP tasks of *TeraGen* took on average 62,420 ms to complete. Compared to the average value 62,113 ms in Test 1 (where x = 8 in

[3]A box-and-whisker contains the following statistics: minimal, maximum, median, the median of the first half, and the median of the second half. The red spot within each box-and-whisker shows the average value.



(A) *TeraGen*



(B) *Pi*

FIGURE 3.3: Average MAP task time in each test (time in ms, lower is better; x axis shows the total number of parallel tasks in each test)

FIGURE 3.3a), the change was negligible, well under 1%. This result was understandable since *TeraGen* is I/O intensive, thus less likely to be affected by CPU resources.

FIGURE 3.3b shows the task performance of *Pi*. The average MAP time in Test 4 (where x = 11) is not significantly longer than in Test 1, which verifies my surmise that CPU was not yet overstressed by 11 tasks[4]. This result was somewhat contrary to the assumption of YARN, according to which the CPU resources were already saturated by 8 tasks in Test 1.

[4]I attributed this slight performance degradation to the overhead, such as context switch, of running more tasks in parallel.

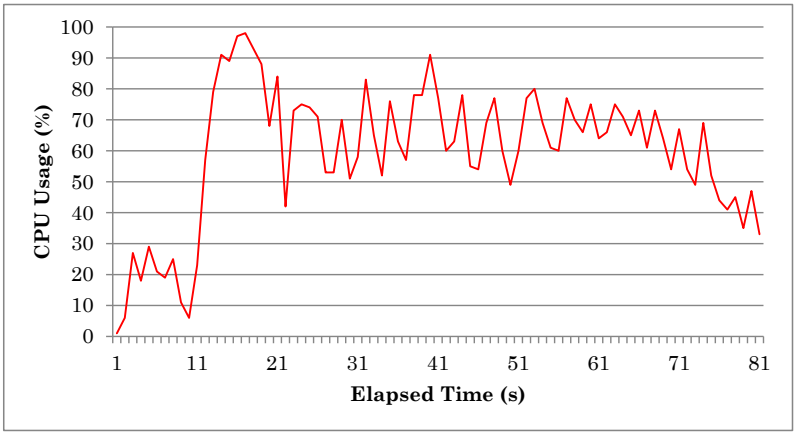


FIGURE 3.4: CPU usage of the datanode in Test 0

To give a more direct view of CPU under-utilisation, I performed Test 0 which executes a *TeraGen* job that contains 7 MAP tasks, each of which generates 512 MB of data. As shown in FIGURE 3.4, the CPU of the datanode was hardly saturated by the 8 tasks (7 MAP tasks and the AM), with an average load of scarcely 60%. In terms of physical CPU cores, only approximately 5 cores were actually saturated. Since MapReduce itself does not have a mechanism to distinguish between CPU- and I/O- intensive workloads, some datanodes could end up being assigned many I/O-intensive tasks, which could be disastrous, since the scheduler would still assume that the CPU resources of those datanodes are saturated.

The results of the tests above imply that the CPU scheduling of YARN indeed causes under-utilisation. In particular, the assumption that every task saturates one CPU core breaks under workload patterns that contain I/O intensive jobs. To solve this problem, the actual CPU resource requirement of workloads have to be taken into consideration.

3.2 Finer Grained CPU Resource Model

The memory allocation in YARN gives an excellent example of flexible and fine grained resource management

1. Each datanode has an amount of memory dedicated to MapReduce tasks
2. Each task requests a configurable amount based on its need

3. In practice, the memory request for a task is slightly larger than its actual requirement, so that the task would not fail due to insufficient memory

Most of these could be perfectly emulated by the CPU scheduling. Although I have a different reason for 3. By slightly over-requesting CPU resources for a each task, the actual CPU usage in the cluster is unlikely to exceed the capacity, therefore this strategy could effectively avoid over-stressing the CPU resources.

In this section, I describe my CPU resource model that parametrises the CPU resources, and adopts an analysing-based approach to learn the requirements of various workloads. For the remainder of the thesis, if not otherwise specified, J refers to a MapReduce job, T refers to a task, W refers to a workload, and R refers to the CPU resource requirement of a job/task/workload. In addition, for the purpose of clarification, the following conventions apply

1. For MapReduce job J, its
   (a) **Workload** W_J refers to the definition of the job. In MapReduce, the behaviour of the job is specified by its MAP and REDUCE tasks, or more specifically, by its underlying Mapper and Reducer class definitions. In programming language terms, W_J could be viewed as a class, and J could be treated as a specific object of W_J. Two jobs belong to the same workload iff they have the exact same Mapper and Reducer class
   (b) **Resource Requirement** refers to all the information about its: memory requirement per MAP task; memory requirement per REDUCE task[5]; CPU requirement per MAP task; CPU requirement per REDUCE task. All these should be dynamically configurable before J is submitted to the RM. In particular, the **CPU Requirement** R_J refers to the CPU-related configurations
2. For task T
   (a) J_T refers to the MapReduce job that contains T
   (b) Its **Workload** W_T refers to the workload of J_T

[5]The per-REDUCE-task configurations apply iff the job contains any REDUCE tasks.

3.2.1 Parametrisation

The first question that arises in my resource model is: **Q1. How should the CPU resources be parametrised?** Clearly, this parametrisation should be able to quantify the node capabilities and the task requirements in unified units. While memory resources could simply be expressed as multiples of MB/GB/..., there does not exist a natural way to do so for CPU resources[6].

Since CPU consumption in practice is often expressed as percentage, I define the capacity of each physical CPU core as 100, representing 100% of its processing power. For each datanode, given the number of physical cores n, its CPU resource capacity is

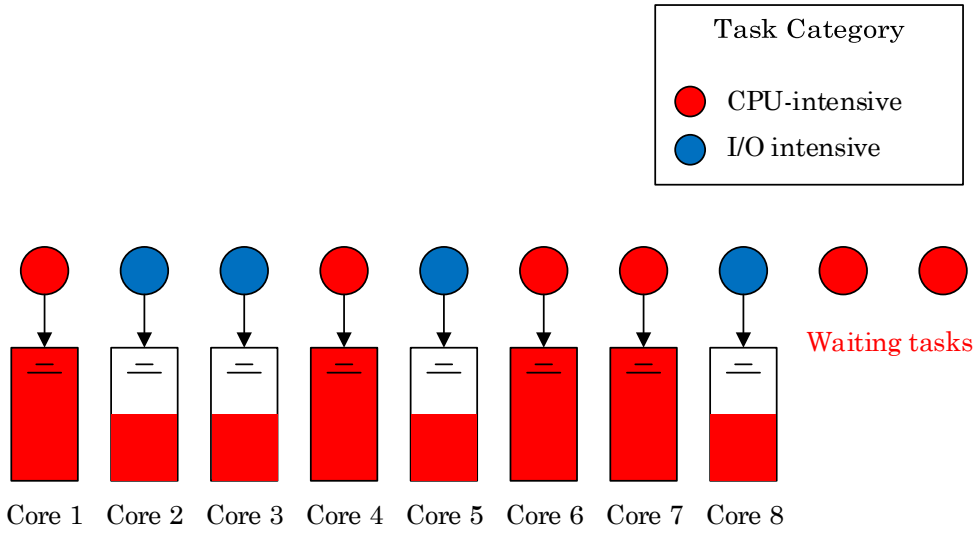
$$Capacity = n * 100 \tag{3.1}$$

For task T, its CPU resource requirement R_T, which is an integer ranging from 1 to 100, has the meaning that "T saturates R_T% of one physical core". The CPU allocation policy of YARN could thus be expressed as "every task requires 100 CPU resources, i.e., it always saturates one core". I deprecate that assumption with the exception of the following two cases

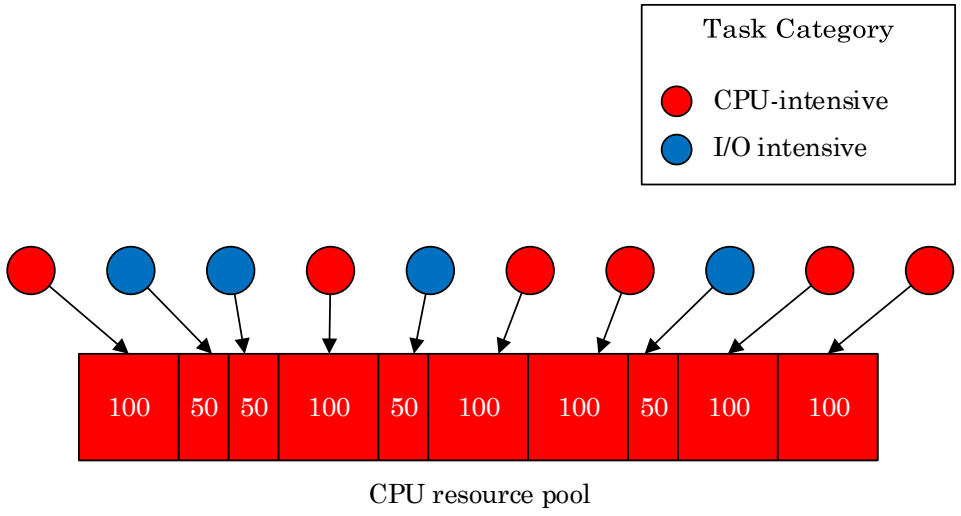
- The workload W_T is "unknown". As described in Section 3.2.2, I adopt an analysing based approach to obtain the CPU resource requirement of W_T. Consequently, W_T remains unknown until the analysis is completed
- T is too small. In practice, there are often tasks that finish in scant few seconds. In this case, it is difficult to ensure the accuracy of the CPU requirement analysis, as the overhead of task execution becomes non-negligible. In addition, it is not worth the effort to concern over a very small task - it would finish quickly and free up the CPU resources anyway

In either case, 100 CPU resources are requested for T by default. This is tantamount to assuming that T would saturate one physical core, but it is necessary to do so, as we do not know the amount of CPU resources that T actually requires. Using the assumption of YARN might cause under-utilisation, but at least could effectively avoid over-stressing. Fortunately, the first case occurs only when the

[6]YARN models CPU resources in terms of cores, but as discussed in Sections 3.1, this lacks flexibility and efficiency, and is exactly what I are trying to be rid of



(A) YARN



(B) Finer Grained

FIGURE 3.5: CPU scheduling on a datanode: YARN vs. Finder Grained

first job of W_T is executed, thus very infrequent[7]; the second case does not have significant impact on the overall utilisation since the task is very short.

I refer to the above parametrisation as "Finer Grained". The name is self-explanatory. In my approach, the CPU resources are no longer viewed by the scheduler as individual cores. Instead, each datanode is configured with a "pool of CPU resources", the capacity of which is defined by Equation 3.1. A task T could then request a configurable amount, based on its requirement R_T.

[7]In practice, we tend to execute the same workload again and again, on different inputs.

FIGURE 3.5 illustrates the CPU scheduling on a single-node cluster. Assume that the datanode has 8 physical cores; the cluster is mixed with CPU- and I/O-intensive workloads; the CPU resource requirement of each CPU-intensive and I/O intensive task is 100 and 50 respectively. As shown in FIGURE 3.5a, YARN allows at most 8 parallel tasks, based on its assumption that each requires a whole core. FIGURE 3.5b illustrates the Finer Grained CPU policy, which views the CPU resources as a pool rather than individual cores, and makes scheduling decisions according to the task requirements. As a result, it enables more tasks in parallel. Although the disk utilisation is less likely to be significantly improved by more parallel I/O-intensive tasks[8], the Finer Grained scheduling takes full advantage of the wasted proportion of CPU resources, thus effectively improving the utilisation.

3.2.2 CPU Requirement Analysis

With CPU resources parametrised in Section 3.2.1, the second question arises: **Q2. For a particular task T, how to calculate its CPU resource requirement R_T?**

In my model, R_T is estimated to be the "CPU intensity" of T. This CPU intensity is opposed to I/O intensity, and is calculated as follows

$$R_T = \frac{CPU\ Time}{CPU\ Time + I/O\ Time} \times 100 \tag{3.2}$$

The intuition is that the more CPU-intensive T is, the more time T spends on CPU computations compared to I/O operations, thus the higher the value of R_T, e.g., if T is performing computations only throughout the entire course, R_T is estimated to be 100, meaning "T saturates one core".

In Equation 3.2, the CPU time is relatively stable as it is roughly proportional to the amount of calculations, which is almost fixed for a particular task. It is also feasible to get the relatively accurate CPU time of the task JVM[9]. On the contrary, it is difficult to directly get the I/O time. A good indication seems to be

$$I/O\ Time = Wall\ Clock\ Time - CPU\ Time \tag{3.3}$$

[8] As mentioned at the beginning of the chapter in page 19, running even one I/O-intensive task would nearly saturate the disk bandwidth.

[9] The robust way is to get the process id of the task JVM, and then get the CPU time via "/proc/[pid]/stat". It is used by many Hadoop components.

Using Equation 3.3 would substitute the denominator in Equation 3.2 with the wall clock time of T. However, had R_T been calculated using Equation 3.2 and 3.3, it would have violated a vital principle of my design - avoid over-stressing the CPU resources. If a CPU-intensive job J had suffered uncommonly poor I/O performance due to fierce contentions, its wall clock time would have been longer than it should normally be, causing its CPU requirement to be underestimated. Consequently, the scheduler would end up assigning more CPU-intensive tasks than it ought to, potentially over-stressing the CPU of some datanodes.

To stick to the principle, I calculate $I/O\ Time$ in Equation 3.2 based on the total number of bytes read and written by T and the I/O rate of the datanodes. Moreover, I pretend that T always had optimal I/O performance, i.e., the full disk bandwidth of the relevant datanode. Typical implementations of MapReduce such as Hadoop usually run on non-overlapping FileSystem[10] and HDFS, therefore $I/O\ Time$ could be calculated as follows

$$I/O\ Time = \frac{FileSystem\ Bytes\ Read}{FileSystem\ Read\ Rate} + \frac{FileSystem\ Bytes\ Written}{FileSystem\ Write\ Rate} + \frac{HDFS\ Bytes\ Read}{HDFS\ Read\ Rate} + \frac{HDFS\ Bytes\ Written}{HDFS\ Write\ Rate} \tag{3.4}$$

and each task could be calculated by Equation 3.2 and 3.4.

In MapReduce, jobs have a-priori unknown resource requirements, therefore my last question is: **Q3. How do I know the CPU resource requirement of a new job?**

As Ren et al. [19] reported in their measure study on three Hadoop clusters for research, the bulk of the use of Hadoop is in "repetitive transformations", i.e., the same workload tends to be repeatedly executed, on different inputs. Therefore for each workload W, its CPU requirement R_W could be analysed by executing a sample job that contains a small number of tasks; once the analysis is completed, future jobs of W could all benefit from it, thus the overhead of the small sample job could be amortised over time.

For a sample job J, the CPU requirement per MAP/REDUCE task could be estimated to be the average among all its MAP/REDUCE tasks. This is appropriate due to the similarity amongst tasks in the same job[11]. To obtain the FileSystem

[10] FileSystem refers to the local file system of the datanode on which the task is executed.
[11] As indicated by Gupta et al. [12] in their ThroughputScheduler.

and HDFS rate for Equation 3.4, a series of I/O tests need to be conducted. While testing I/O rates, it is important to eliminate the factor of memory buffer cache. Details on this matter are provided in Section 3.3.

Suppose J contains MAP tasks T_{M1}, T_{M2}, ..., T_{Mm}, and REDUCE tasks T_{R1}, T_{R2}, ..., T_{Rn}, the CPU requirement of workload W_J could be calculated as follows

$$R_{W_J}^{MAP} = \frac{\sum_{i=1}^{m} R_{T_{Mi}}}{m} \tag{3.5a}$$

$$R_{W_J}^{RED} = \frac{\sum_{j=1}^{n} R_{T_{Rj}}}{n} \tag{3.5b}$$

where $R_{W_J}^{MAP}$ and $R_{W_J}^{RED}$ represent the CPU resource requirement per MAP task and per REDUCE task respectively. Equation 3.5b is ignored if W_J is MAP-only.

There are various considerations when analysing the CPU resource requirements in practice. They are to be discussed in Section 3.3.

3.3 Implementation

This section presents a practical scheduling system that makes use of the resource model proposed in Section 3.2. I firstly outline the system design, then provide a few details of my implementations. To avoid reinventing the wheel, I build my scheduler on top of Hadoop YARN due to already advanced memory management.

3.3.1 System Design

To analyse the CPU resource requirement of a workload, firstly I should to be able to analyse individual tasks. To calculate a particular task, firstly the information on the FileSystem and HDFS I/O rate is necessary. Those I/O rates could be obtained by conducting a series of I/O tests. While on the tests, it is important to eliminate the factor of memory buffer cache, i.e., it is vital to ensure that

For reading tests, the data is actually read from disk

This could be done by writing a dummy file larger than the size of the memory before each test to flush out the buffer cache, so that later in the actual test, the data is directly read from disk

For writing tests, all the data is written to disk

For local FileSystem, a call of **fsync()** could provide this guarantee. As for HDFS, since currently none of the stable releases of Hadoop implements **hsync()** as its real expected semantics, which should be "posix fsync equivalent" [20], a workaround is to switch on the "dfs.datanode.sync.behind.writes" configuration[12] so that the datanode would instruct the operating system to enqueue all written data to the disk immediately after it is written [3]

To analyse a particular task T using Equation 3.2 and 3.4, five more values are essential - the total CPU time; the total number of FileSystem bytes read; the total number of FileSystem bytes written; the total number of HDFS bytes read; the total number of HDFS bytes written. In practice, task execution incurs overhead, but I want to focus on the task itself. Therefore, I calculate the five values in two steps

- **Obtain Starting Values** I obtain the values of the five parameters at the very beginning of T, ideally at the point when T has not performed any actions, and store them as the starting values
- **Calculate Totals** At the very end of T, ideally when T has finished all its operations and is just about to exit, I obtain the values again. By subtracting the starting values, the total CPU time spent by T, the total number of FileSystem bytes read by T, etc., could all be calculated

For each workload W, I analyse its CPU resource requirement by executing a sample job J that contains a small number of tasks. Once all tasks finish, they are put together to calculate the average, as described in Equation 3.5. Assume the two calculated values are R_W^{MAP} and R_W^{RED}, then future jobs of W could configure their per MAP CPU requirement as R_W^{MAP}, and per REDUCE requirement as R_W^{RED}, respectively. When analysing workload W with sample job J, a few extra considerations are given to ensure the reliability of the results

- If a particular task T has more than one attempts, only the successful attempt is involved in the final calculations of R_W in Equation 3.5. In MapReduce, T could have multiple task attempts if: (a) the initial attempt(s) failed;

[12]This feature is normally used as an optimisation for some workloads by smoothing the I/O operations. HDFS-related properties could be set in the "hdfs-site.xml" configuration file.

(b) T was speculatively executed[13]. Speculative execution is off in my experiments, but it is a useful feature and might be desired elsewhere. Since my system is meant for practical use, I do consider the possible impact of speculative executions

- The analysis results are written for future use iff J is successful. It makes sense not to trust the statistics of a failed job

3.3.2 Scheduling Framework

This section describes the implementation of Finer Grained CPU scheduling. A rule of thumb in my design is to preserve the simplicity of MapReduce, as described in Section 1.2. Ideally, I want to avoid asking for any extra effort from the Hadoop user, but that is difficult in practice as my system requires at least testing I/O rates. Fortunately, those tests only need to be conducted once every time the disk hardware in the cluster changes, which is rather infrequent. The tested I/O rates could be placed in one of the built-in XML configuration[14] files so that they could be conveniently loaded.

To avoid tedious manual configurations, I build my features into Hadoop framework so that the user could write his jobs as usual, leaving the cumbersome CPU requirement analysis to Hadoop. In the following subsections, I briefly review the work-flow of a MapReduce job. In each step, I firstly present my objective, and then describe relevant implementations. I strive to make my system portable, i.e., independent of any specific Hadoop release. FIGURE 3.6 shows the simplified workflow, with Finer Grained CPU Scheduling built in. To distinguish with the Hadoop framework, I use a different colour for my features.

3.3.2.1 Job Submission

To start a MapReduce job J, J needs to be configured and submitted to the RM. During the configurations, the CPU resource requirement per MAP/REDUCE task could be specified. As described in Section 3.3.1, I analyse workloads by executing sample jobs, and store the results for future scheduling. Therefore,

[13] More explanations on speculative execution are given in footnote 2 on page 12.

[14] Hadoop includes 4 built-in configurations XML files: "core-site.xml", "hdfs-site.xml", "mapred-site.xml", and "yarn-site.xml" [3].

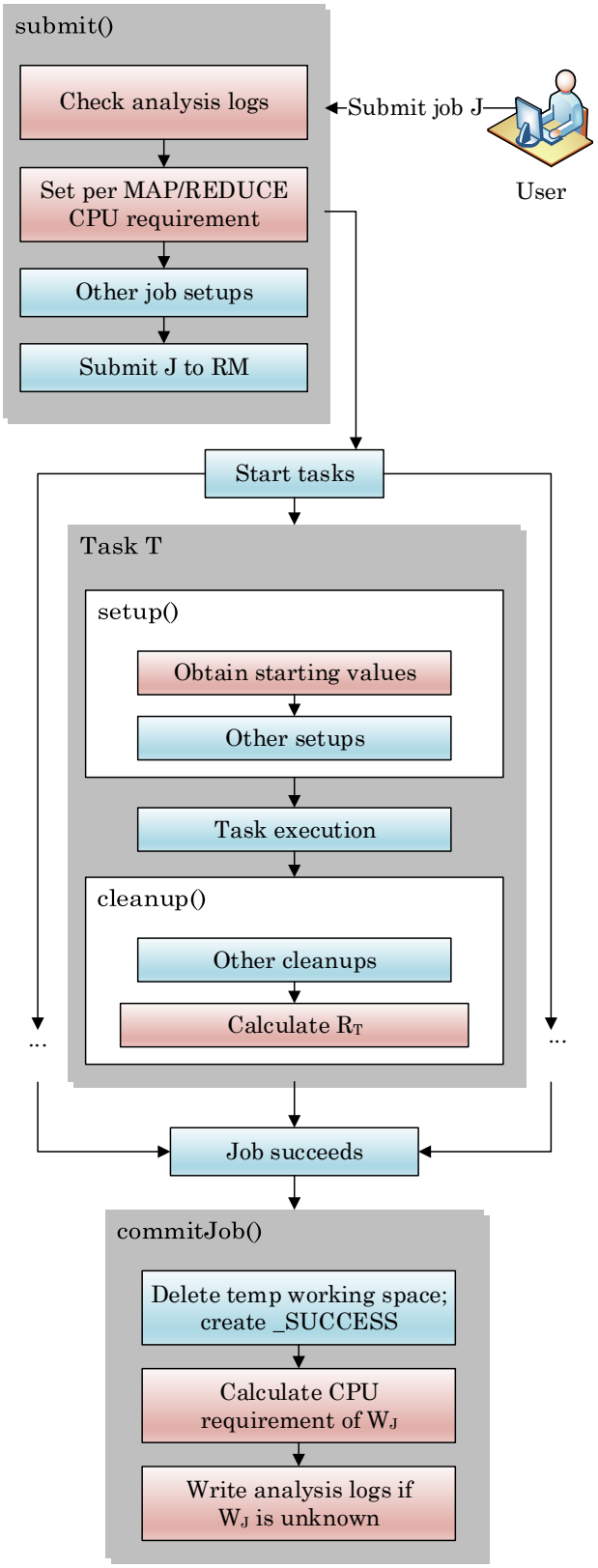


FIGURE 3.6: Simplified workflow of Finer Grained CPU scheduling

if the workload of J is "known", i.e., W_J has already been analysed, J could configure its per task CPU requirement according to the previous analysis results. Otherwise, J acts like a sample job, and its per task CPU resource requirement is set to the default 100. As described in Section 3.2, this default setting is a mechanism to avoid over-stressing the CPU resources in the cluster.

In Hadoop, there are two interfaces for submitting job J

- **submit()** submits J and returns immediately
- **waitForCompletion()** submits J and polls for progress (if the user desires so) until J is complete, then returns

Both methods eventually go through the **submit()** method, at the beginning of which J has not been actually submitted, thus extra configurations could be made. To enable Finer Grained CPU scheduling, I check the HDFS folder *cpustat* where the analysis results are stored, and configure the CPU resource requirement if there is a log for W_J; otherwise, this step is skipped and J keeps the default setting.

3.3.2.2 Task execution

When job J is accepted by the RM, its AM is launched, and the task execution begins. As described in Section 3.3.1, to analyse a particular task T, I need to obtain the values of five parameters, one time at the very beginning of T, another time at the very end. Once R_T is calculated using Equation 3.2 and 3.4, the value needs to be stored in a robust and efficient way[15], so that when J completes, all its tasks could be put together to calculate the job requirement R_J.

In MapReduce, every MAP/REDUCE task is defined by a **Mapper/Reducer** class, which eventually inherits a base **Mapper/Reducer** class[16]. For the purpose of clarification, I refer to those two base classes as **MapperBase** and **ReducerBase** respectively for the remainder of this thesis. The features for calculating R_T

[15]Storing R_T in a local file does not meet the requirement of putting together tasks scattered across the cluster, and using an HDFS file is by no means efficient - the overhead of connecting to the datanode, opening and closing file, etc. is very likely more significant than the cost of the I/O operation itself.

[16]The user could choose to write a **Mapper/Reducer** from scratch, but it would be unwise to reinvent the wheel.

are implemented within **MapperBase** and **ReducerBase** so that they would be inherited to every MAP/REDUCE task.

Section 3.3.1 indicates when to obtain the values of five parameters for calculating R_T. In MapReduce, before the actual execution of T starts, the **setup()** method is called for task set-ups. When T finishes, it invokes **cleanup()** before exit. The very beginning of **setup()** and the very end of **cleanup()** are the exact two points when I need to obtain the values.

Once R_T is calculated using Equation 3.2 and 3.4, I store the value robustly and efficiently via Hadoop counters. I keep counters Map_Num / Red_Num for the total number of MAP / REDUCE tasks, and Map_Cpu / Red_Cpu for the total CPU requirements of MAP / REDUCE tasks respectively, so that when all tasks finish and counters are aggregated, I would have everything needed to analyse W_J using Equation 3.5.

To ensure the reliability of the calculation results, I take the following measures

- Increment counters iff T is successful
- Increment the right counters. Careful considerations are given on this matter due to the commonly adopted optimisation *Combiner*, which performs "local" REDUCE-like functions within MAP tasks. In **ReducerBase**, I insert statements before incrementing counters to ensure that Red_Num and Red_Cpu are incremented only by a true REDUCE task, not a Combiner within a MAP task

The workflow of a task T, with my analysing features built-in, is as follows

```
1: function SETUP(context)
2:     startCPUTime ← current CPU time of task JVM
3:     Obtain startFSRead, startFSWrite, startHDFSRead, startHDFSWrite
4:     Other task set-ups (optional)
5: end function
6: for all split in inputSplits do
7:     if T is MAP task then
8:         Do map() on split
9:     end if
10:    if T is REDUCE task then
```

```
        Do reduce() on split
    end if
end for
function CLEANUP(context)
    Other task clean-ups (optional)
    totalCPUTime ← the CPU time of task JVM - startCPUTime
    Calculate totalFSRead, totalFSWrite, totalHDFSRead, totalHDFSWrite
    if T is not successful then
        return
    end if
    if T is MAP task then
        Increment MAP_NUM and MAP_CPU
    end if
    if T is REDUCE task then
        Increment RED_NUM and RED_CPU
    end if
end function
```

3.3.2.3 Job completion

When all tasks of job J finish, the CPU resource requirement of workload W_J could be calculated using Equation 3.5. As described in Section 3.3.1, the analysis results should be written iff J is successful. I provide this guarantee by writing the results at the end of the **commitJob()** method, due to the fact that **commitJob()** is called iff J completed successfully[17].

In addition, before writing results, I check if an analysis for W_J already exists. If that is the case, the newly analysed results are discarded. To re-analyse W_J, the old logs have to be explicitly cleared using my script. This strategy resembles many real-world MapReduce applications - the job does not overwrite existing paths/files.

[17] If J failed, **abortJob()** is called instead.

Chapter 4

Experiments

In this chapter, I conduct a series of experiments to evaluate the Finer Grained CPU Scheduler proposed in Chapter 3. The objective of my work is to effectively improve the utilisation of CPU resources in the cluster. In particular, under the mixed workload pattern, more tasks should be able to run in parallel without over-stressing the CPU resources in the cluster.

As mentioned at the beginning of Chapter 3, at the moment I are not considering the optimisation on the I/O-intensive workloads. Nevertheless, I do pay attention to their performance - the improvement on the CPU-intensive workloads should not come at the price of degraded I/O resource utilisation.

Hardware	
CPU	Xeon E5530, 2.40 GHz, 8 cores
Memory	24 GB
Disk	500 GB
Network	10 Gbps
Software	
Hadoop	2.6.0
Framework	YARN
Java	Oracle 1.7.0_21

Table 4.1: Experimental environment (per datanode)

Block Size	Read Rate	Write Rate
8k	73.8	72.5
16k	73.4	72.6
32k	73.3	72.6
64k	73.8	72.8
128k	73.8	73.1

TABLE 4.2: dd test results (MB/sec)

4.1 Environment

The experiments were conducted in an 8-node cluster. The hardware specifications of each datanode are summarised in TABLE 4.1. Each of my datanode has 8 physical CPU cores, which according to the resource model of YARN limits the number of parallel tasks per node to 8, but by my standard defines the per-node CPU resource capacity as $8 * 100$.

In practice, the Hadoop daemons on each datanode do consume CPU resources, especially when HDFS is performing a lot of I/O operations, the relevant DataNode processes could have very high CPU usage[1]. Due to this consideration, on each datanode I reserve one physical core for Hadoop daemons and the operating system, leaving a 7*100 per-node CPU capacity for MapReduce tasks. This strategy resembles memory allocation in YARN: it is good practice to reserve a proportion of memory for the system on each node and leave the rest for actual tasks.

I performed a group of tests to obtain the I/O rates of the FileSystem

1. **dd test** [21] TABLE 4.2 summarises the test results under different block sizes. The block size actually had little impact on the performance

   (a) I firstly used the dd command to write 120 GB of data, which was much larger than the memory size (24 GB) so that the buffer would have little effect; in addition, I set the *fdatasync* flag to do a "complete" sync once before exit, so that all the data was actually written to the disk before the rate was calculated

   (b) Next I wrote a large dummy file, which was of size of the memory, to flush out the buffer

[1] I observed 100% load on one CPU core in my environment.

(c) Finally I used the dd command to read the file written in step (a). Since I did (b), I had the guarantee that the data was directly from the disk

2. **C I/O test** This was in essence an emulation of the dd test. The only difference was that in step (a), I did a POSIX **fsync()** to force a sync. The results were rather consistent with the dd test as well, ranging between 73.1 MB/sec and 73.8 MB/sec for both reading and writing

As emphasized in Section 3.2, a vital principle of my scheduler is to avoid over-stressing the cluster CPU resources. In practice, I provide this guarantee by always using the optimal I/O rates in Equation 3.4. As such, for the FileSystem in my environment, I take the ceiling value of the test results

$$FileSystem\ Read\ Rate = 74MB/sec \tag{4.1a}$$

$$FileSystem\ Write\ Rate = 74MB/sec \tag{4.1b}$$

The HDFS I/O rates were tested using the Java APIs in the same fashion. I only give the results here

$$HDFS\ Read\ Rate = 65MB/sec \tag{4.2a}$$

$$HDFS\ Write\ Rate = 65MB/sec \tag{4.2b}$$

4.2 Benchmarks

To evaluate the Finer Grained CPU Scheduler under the mixed workload pattern, I chose *Pi* and *Bbp* as the CPU-intensive workloads, and *TeraGen* as the representative I/O-intensive workload [18]. In addition, I built my own I/O-intensive benchmark *ETL*. The details for each workload is as follows

- ***Pi*** The program that estimates π using the Quasi-Monte Carlo method[2]
- ***Bbp*** This job computes x exact digits of π using the Bailey-Borwein-Plouffe formula. x is configurable, and the calculations are evenly distributed to all MAP tasks

[2]This is to be distinguished with the package org.apache.hadoop.examples.pi, which computes π using the distributed Bailey-Borwein-Plouffe method.

- ***TeraGen*** In this job, each MAP task is specified to write 2 GB of random data to HDFS
- ***ETL*** I constructed this highly I/O intensive workload. Each MAP task reads an HDFS block of plain text, converts it to XML, and writes the result back to HDFS. This resembles many real-world Expand-Transform-Load applications used for preprocessing in Big Data

I conducted 3 groups of tests, which resemble 3 real-world workload patterns

1. **Balanced Pattern** A CPU-intensive job runs along with an I/O-intensive job. Each CPU-intensive job contains 70 MAP tasks; each MAP task of *Pi* processes 2 billion points; *Bbp* calculates 180,000 digits in total. Each I/O-intensive job contains 31 tasks, so that under the CPU policy of YARN, it would roughly take half of the cluster with its 31 MAP tasks and the AM (the whole cluster is able to run 64 parallel tasks under YARN)
2. **CPU-heavy Pattern** Each test contains 3 jobs: *Pi*, *Bbp* and one of the I/O-intensive workloads. Each CPU-intensive job contains 50 tasks; each MAP task of *Pi* processes 2 billion points; *Bbp* calculates 150,000 digits in total. Each I/O-intensive job contains 20 tasks
3. **I/O-heavy Pattern** Each test contains 3 jobs: one of the CPU-intensive workloads, *TeraGen*, and *ETL*. Each CPU-intensive job contains 50 tasks; each MAP task of *Pi* processes 2.5 billion points; *Bbp* calculates 180,000 digits in total. Each I/O-intensive job contains 20 tasks

I tuned the input sizes of jobs in each pattern such that under the scheduling policy of YARN, jobs in each test would have similar execution times (it would be unfair to assess the performance of a very large job and a tiny one, since the latter would have little impact). For each test, I would repeat for 10 times to calculate the average, standard deviation, etc.

Before proceeding to the performance evaluation, I made use of the features described in Section 3.2 to analyse CPU resource requirement of the workloads. Each workload was analysed with a sample job that contained 16 tasks.

When running sample jobs, I stuck to the default per-task CPU request 100, since at this point workloads were "unknown". In addition, some tasks are extremely

Workload	MAP	REDUCE
Pi	100	Default
Bbp	100	100
TeraGen	49	N/A
ETL	47	N/A

TABLE 4.3: Per MAP/REDUCE task CPU requirement of each workload

short-lived, e.g., *Pi* contains a single REDUCE task that finishes in scant few seconds. For this kind of tasks, it is difficult to guarantee the accuracy of resource requirement analysis as the overhead of task execution becomes non-negligible. Thus I ignored the analysis results and changed them to "default". As explained in Section 3.2.1, this default setting is a safe option to avoid over-stressing the cluster CPU resources.

The CPU requirements of the workloads are summarised in TABLE 4.3. For MAP-only jobs, the REDUCE column is left N/A.

4.3 Metrics

I assess the performance of each job in two aspects

- **Job execution time** The objective of Finer Grained CPU scheduling is to make the most of CPU resources to enable more tasks in parallel. Although more parallel I/O-intensive tasks are unlikely to significantly improve the disk utilisation, CPU-intensive jobs could benefit from higher parallelism without their individual performance being compromised, due to my vital principle that avoids over-stressing the CPU resources. As a result, I expect a reduction on the execution time of CPU-intensive jobs, and unaffected performance on the I/O-intensive jobs
- **Average MAP task time** The caveat of higher parallelism is the potentiality of over-stressing the cluster resources, which would severely compromise per-task performance. Therefore, I assess the average execution time of MAP tasks in each job as an indicator of task performance. All the workloads have very small REDUCE tasks or none, thus comparing their performance is meaningless in my experiments

Test	Workload	YARN	Finer Grained	Degradation
Pi+TeraGen	Pi	78,186	78,066	N/A
	TeraGen	196,997	177,502	N/A
Pi+ETL	Pi	78,563	78,039	N/A
	ETL	225,515	198,143	N/A
Bbp+TeraGen	Bbp	90,937	90,981	0.05%
	TeraGen	220,322	212,706	N/A
Bbp+ETL	Bbp	90,489	90,875	0.04%
	ETL	271,150	233,420	N/A

TABLE 4.4: Average MAP task time (in ms, lower is better) under balanced workload pattern

4.4 Results

I compared my Finer Grained CPU Scheduler with the state-of-art YARN scheduler. Since I was running mixed workloads, there needed to be a policy for memory sharing amongst jobs. I chose Fair Scheduler since it allows jobs in each test to progress simultaneously.

4.4.1 Results under Balanced Pattern

In each test I selected one CPU-intensive job and an I/O-intensive job, and executed them together, therefore under the balanced workload pattern there were 4 different tests.

Firstly, compared to YARN, the Finer Grained CPU Scheduling does not degrade the individual task performance in any test. As shown in the "degradation" column in TABLE 4.4, in most cases the task performance is even slightly better than YARN (marked as N/A since there is no degradation); even when Finer Grained does increase the task time, the change is well under 1%, therefore completely negligible.

Secondly, as shown in FIGURE 4.1[3], under Finer Grained CPU scheduling, the execution time of the CPU-intensive workload in every test was significantly reduced, by 18% up to 23%. This improvement is rather understandable since the Finer Grained approach enables more CPU-intensive tasks in parallel yet does not

[3]In each of the graphs, red colour and blue colour represent the CPU- and I/O-intensive job(s) respectively. The solid fill shows the results under YARN, while the pattered fill shows the results under Finer Grained CPU Scheduling.

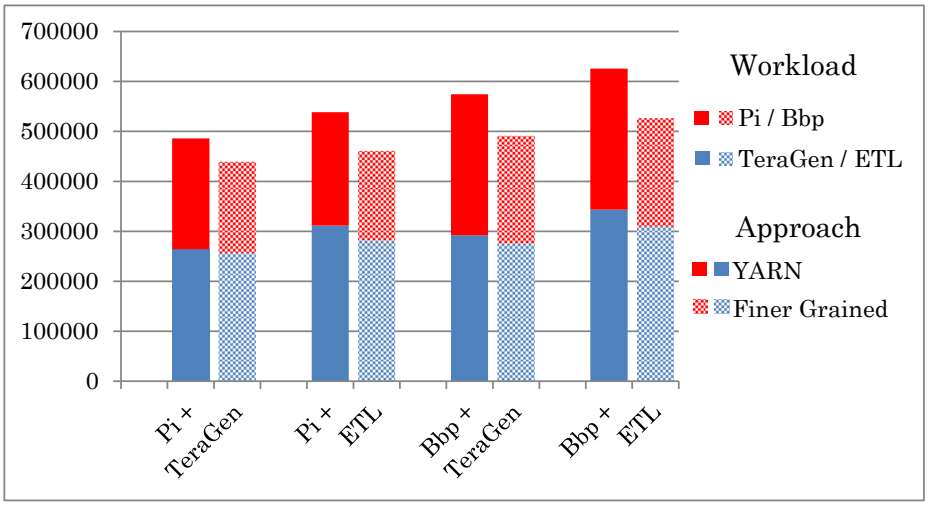


FIGURE 4.1: Job execution times (in ms, lower is better) under balanced workload pattern

Test	Workload	YARN	Finer Grained	Degradation
Pi+Bbp+TeraGen	Pi	78294	78224	N/A
	Bbp	87,525	87,585	0.07 %
	TeraGen	98,731	94,548	N/A
Pi+Bbp+ETL	Pi	77,822	78,179	0.46%
	Bbp	86,890	87,399	0.59%
	ETL	117685.9	110685.664	N/A

TABLE 4.5: Average MAP task time (in ms, lower is better) under CPU-heavy workload pattern

over-stress the CPU resources, therefore does not compromise the individual task performance. In addition, the execution time of I/O-intensive jobs were not affected. This verifies that the improvement on the CPU utilisation does not come at the price degraded I/O resource utilisation.

4.4.2 Results under CPU-heavy Pattern

Under such pattern, in each test I selected one I/O-intensive workload, and ran it along with both CPU-intensive jobs, therefore there were 2 different tests: $Pi + Bbp + TeraGen$ and $Pi + Bbp + ETL$.

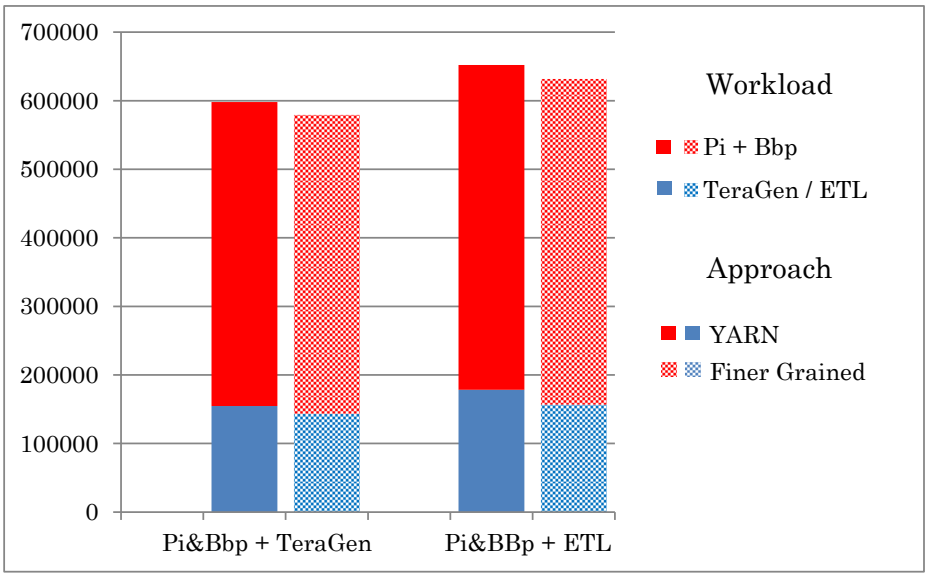


FIGURE 4.2: Job execution times (in ms, lower is better) under CPU-heavy workload pattern

Test	Workload	YARN	Finer Grained	Degradation
Pi+TeraGen+ETL	Pi	95,941	95,761	N/A
	TeraGen	177,151	176,626	0.07 %
	ETL	194,363	192,373	N/A
Bbp+TeraGen+ETL	Bbp	86,114	87,932	2.1%
	TeraGen	178,407	175,259	N/A
	ETL	192,785	189,189	N/A

TABLE 4.6: Average MAP task time (in ms, lower is better) under I/O-heavy workload pattern

Firstly, as shown in TABLE 4.5, there is no performance degradation in individual tasks. However, the job performance is almost identical compared to YARN, as shown in FIGURE 4.2. The reason is that under such workload pattern, i.e., where most of the tasks are CPU-intensive, the proportion of CPU resources wasted by YARN is very limited, therefore there is not so much room for improvement.

4.4.3 Results under I/O-heavy Pattern

There were 2 different tests under this pattern: $Pi + TeraGen + ETL$ and $Bbp + TeraGen + ETL$.

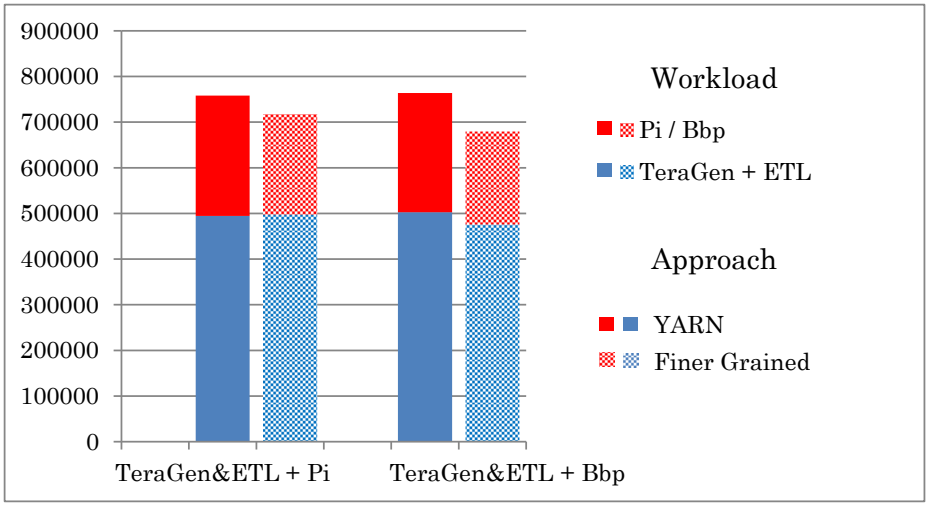


FIGURE 4.3: Job execution times (in ms, lower is better) under I/O-heavy workload pattern

TABLE 4.6 shows the average MAP task time. Although *Bbp* in the second test suffers a slight degradation (2.1%), the overall task performance under Finer Grained is not worse than YARN.

The improvements on the job performance, as shown in FIGURE 4.3, is consistent with the results under balanced workload pattern. I/O-intensive jobs are not compromised, and the execution time of CPU-intensive jobs in two tests are reduced by 17% and 22%, respectively.

To sum up the 3 groups of tests

- Under CPU-heavy workload pattern where most of the tasks indeed saturate one CPU core, the benefit of Finer Grained CPU Scheduling is limited. Nonetheless, it does not degrade the resource utilisation or the job performance
- Under balanced and I/O-heavy pattern, the CPU assumption of YARN causes under-utilisation, therefore the Finer Grained approach becomes very useful. It effectively increases the CPU utilisation and therefore the job performance of CPU-intensive workloads, without compromising the I/O utilisation in the cluster

Chapter 5

Conclusion

Many commonly used scheduling methods for MapReduce work on the resource sharing policy amongst jobs, or more specifically, the decision making strategy on the selection of job to schedule resources for at any particular moment such that jobs achieve, on average, better performance. For a cluster with a particular resource capacity, improving the overall performance of jobs running in it is also beneficial to the resource utilisation.

Strictly speaking, however, resources are allocated to tasks rather than directly to jobs in MapReduce. Consequently, much could be done on the task level to further improve the resource utilisation, yet such work is relatively lacking. In this thesis, I discussed the efficiency of resource allocation in two aspects

A1. Avoid over-stressing but alleviate waste
This is what YARN has been trying to achieve in its refined resource model. By taking into account the memory requirement of each task, memory utilisation in YARN is significantly improved

A2. Consider various workload patterns
In practice, there are CPU- and I/O- intensive workloads. However, YARN assumes the same CPU requirement for every task, and the logic of ThroughputScheduler does not apply to a homogeneous environment

As such, I proposed a Finer Grained CPU Scheduler, which alleviates the problem of CPU resource waste in the state-of-art approach YARN. In particular, by estimating the CPU requirement of each task, CPU- and I/O- intensive workloads are

differentiated, and the wasted proportion of CPU resources are effectively utilised. In addition, I avoid over-stressing the CPU resources by slightly over-estimating the task requirements, like YARN does with memory allocation. To sum up, the Finer Grained CPU Scheduler achieves both [**A1**] and [**A2**] as mentioned above.

I always kept in mind that MapReduce is meant for practical use. Therefore I built my scheduler into the Hadoop framework. My CPU scheduling does require a set of I/O tests when the cluster hardware changes, but that is rather infrequent. The cumbersome CPU requirement analysis and allocations are left to the Hadoop framework, and the user could write his jobs as usual, i.e., the simplicity of MapReduce is well preserved.

At present I am not ready to model the disk resources, not are most implementations of MapReduce. Had I defined the I/O capacity like the memory or CPU resources, in practice it would have been over-stressed almost the entire time. In my environment, even a single I/O-intensive task would nearly saturate the disk bandwidth of the relevant datanode. As such, the above-mentioned [**A1**] does not seem appropriate for disk resources. Future extension on the I/O could be based on the idea of "assign resources to where they are needed most". In particular, when assigning disk resources, jobs that are more I/O-intensive could be prioritised. Alternatively, the priority could be based on the remaining size of I/O operations of each job.

Bibliography

[1] KV Shvachko and AC Murthy. Scaling hadoop to 4000 nodes at yahoo! *world wide web, http://developer. yahoo. net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a. html*, 2008.

[2] Owen O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at: http://sortbenchmark. org/Yahoo-Hadoop. pdf,(May)*, pages 1–3, 2008.

[3] Apache hadoop, . URL http://hadoop.apache.org/.

[4] Tom White. *Hadoop: The Definitive Guide*. 2012.

[5] Martin Rinard. Operating systems lecture notes, August 1998. URL http://people.csail.mit.edu/rinard/osnotes/h6.html.

[6] M. PASTORELLI, A. Barbuzzi, D. Carra, M. Dell'Amico, and P. Michiardi. Hfsp: Size-based scheduling for hadoop. In *Big Data, 2013 IEEE International Conference on*, pages 51–59, Oct 2013.

[7] Edward W. Davis and James H. Patterson. A comparison of heuristic and optimum solutions in resource-constrained project scheduling. *Management Science*, 21(8):944–955, 1975.

[8] Matei Zaharia. Job scheduling with the fair and capacity schedulers. *Hadoop Summit*, 9, 2009.

[9] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

[10] Alan Burns. Preemptive priority-based scheduling: An appropriate engineering approach. In *Advances in Real-Time Systems, chapter 10*, pages 225–248. Prentice Hall, 1994.

[11] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, 2013.

[12] Shekhar Gupta, Christian Fritz, Bob Price, Roger Hoover, Johan De Kleer, and Cees Witteveen. Throughputscheduler: Learning to schedule on heterogeneous hadoop clusters. In *ICAC*, pages 159–165, 2013.

[13] K. Arun Kumar, Vamshi Krishna Konishetty, Kaladhar Voruganti, and G. V. Prabhakara Rao. Cash: Context aware scheduler for hadoop. In *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ICACCI '12, 2012.

[14] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, 2008.

[15] Ibm - what is the hadoop distributed file system (hdfs), . URL http://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/.

[16] Puneet Singh Duggal and Sanchita Paul. Big data analysis: Challenges and solutions. In *International Conference on Cloud, Big Data and Trust 2013, Nov 13-15, RGPV*, 2013.

[17] Typical workloads patterns for hadoop, . URL http://docs.hortonworks.com/HDPDocuments/HDP1/HDP-1.3.7/bk_cluster-planning-guide/content/typical-workloads.html.

[18] Hadoop example jar, . URL https://github.com/apache/hadoop/tree/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples.

[19] Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *Proc. VLDB Endow.*, 6(10), August 2013.

[20] Hadoop jira: Support hsync in hdfs, . URL https://issues.apache.org/jira/browse/HDFS-744.

[21] Use dd to benchmark your disk or cpu, . URL https://romanrm.net/dd-benchmark.

Publications

1. Kun Liu, Daisaku Yokoyama, Masashi Toyoda, Masaru Kitsuregawa. An Improvement on Hadoop Scheduling by Utilising Analysed CPU Resource Demands. The 7th Forum on Data Engineering and Information Management (DEIM 2015). (to appear)