

國立交通大學

資訊科學與工程研究所

碩 士 論 文

異質 MapReduce 環境中的負載感知排程器

A Load-Aware Scheduler for MapReduce Framework in
Heterogeneous Environments

研 究 生：尤信翰

指 導 教 授：黃俊龍 教授

中 華 民 國 九 十 九 年 八 月

異質 MapReduce 環境中的負載感知排程器
A Load-Aware Scheduler for MapReduce Framework in Heterogeneous
Environments

研究生：尤信翰

Student : Hsin-Han You

指導教授：黃俊龍

Advisor : Jiun-Long Huang

國立交通大學
資訊科學與工程研究所
碩士論文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

August 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年八月

異質 MapReduce 環境中的負載感知排程器

學生：尤信翰

指導教授：黃俊龍 博士

國立交通大學資訊科學與工程研究所

摘要

MapReduce 已成為目前處理海量資料的一個利器。不論資料探勘、處理紀錄檔、處理網頁索引及其他需要大量資料處理的科學研究，都可透過 MapReduce 得到好的程式擴展性以及執行效率。MapReduce 為一分散式批次資料處理程式框架，MapReduce 將一個工作分解為許多小的 map 任務以及 reduce 任務。每個工作節點分別完成一些 map 任務及 reduce 任務來完成整個工作。Hadoop MapReduce 是目前最熱門的 MapReduce 開放原始碼實做。Hadoop MapReduce 有一個可抽換的排程介面 TaskScheduler，預設的排程器將工作用先進先出的方法排程。

排程器如何選擇工作，分配 map/reduce 任務會影響 MapReduce 工作的執行效率與整體工作站群的使用率。在實際的服務中，我們發現有許多問題仍需要在排程時考慮以增進效能，如工作節點的動態負載、工作節點的異質性、同時存在許多工作時的任務選擇。我們發現目前的 Hadoop MapReduce 對於這些問題並沒有妥善處理，並且在相關的情況下，整體效能會下降。我們針對這些問題提出了我們的解法，以及在 Hadoop MapReduce 上實做了我們的 Load-Aware 排程器來提昇整體工作效能。我們的實驗可以在一般的多工作情況下，透過避免不必要的備份任務，提昇平均 10% 至 20% 的效能。

關鍵字：雲端運算，Hadoop MapReduce，排程演算法

A Load-Aware Scheduler for MapReduce Framework in Heterogeneous Environments

Student: Hsin-Han You

Advisor: Dr.Jiun-Long Huang

Institutes of Computer Science and Engineering
National Chao Tung University

ABSTRACT

MapReduce is becoming a trendy programming model for large-scale data processing such as data mining, log processing, web indexing and scientific research. MapReduce framework is a batch distributed data processing framework that disassembles a job into smaller map tasks and reduce tasks. In MapReduce framework, master node distributes tasks to worker nodes to complete the whole job. Hadoop MapReduce is the most popular open-source implementation of MapReduce framework. Hadoop MapReduce comes with a pluggable task scheduler interface and a default FIFO job scheduler. Performance of MapReduce jobs and overall cluster utilization rely on how the tasks being assigned and processed. In practice, there are some issues such as dynamic loading, heterogeneity of nodes, multiple job scheduling needs to be taken into account. We find that current Hadoop scheduler suffers from performance degradation due to the above problems. We propose a new scheduler named Load-Aware Scheduler to address these issues, and improve the overall performance and utilization. Experimental results show that we could improve 10% to 20% of utilization on average by avoid unnecessary speculative tasks.

Keywords: Cloud Computing, Hadoop MapReduce, Scheduling algorithm

致謝

首先要感謝黃俊龍教授兩年來的指導，除了對學術研究的指導，教授在研究上給了我很大的自由度，讓我可以任研一時可以自由探索，並在研二迷惘時出手救援。也很感謝我的口試委員們：黃慶育教授、彭文志教授以及楊得年老師，在口試時提供我許多有用的意見及資訊，以及楊得年老師在碩士班期間的指導。

還要感謝實驗室的夥伴們，羅堯學長、士銓學長、振哲學長、壬禾學長、志安學長、在研究上的教學，還有同屆一起奮鬥的大家，還有眾多學弟們。

另外要感謝在系計中工作四年強大工作夥伴們！尤其是 ch Wong, chia hung, lw hsu, liuyh 學長們，都在我的碩士生涯中給我許多指點幫助。travisyd 每個週四晚上的麥當勞團，讓我可以快快樂樂的進食，還讓我參考了許多 LaTeX 語法，省下許多寶貴的時光。

最後要感謝我的家人，以及女朋友麗洧兩年多來的支持。沒有了他們，我是無法順利完成這份論文的。



Contents

摘要	i
Abstract	ii
致謝	iii
Table of contents	iv
List of figures	vi
1 Introduction	1
2 Preliminary	5
2.1 Hadoop Architecture	5
2.1.1 Hadoop MapReduce Scheduling	6
2.1.2 Fair Scheduler and Capacity Scheduler	8
2.2 Related Work	9
2.2.1 LATE Scheduler	9
2.2.2 Heterogeneous Workloads	10
3 Load-Aware Scheduler	13
3.1 Objectives	13
3.2 Proposed Scheme	14
3.2.1 Data Collection Module	15
3.2.2 Task Scheduling Module	16
3.3 Implementation	17



3.4 Discussion	21
4 Performance Evaluation	23
4.1 Unnecessary Speculative Tasks	23
4.2 Dynamic Loading	24
4.3 Impact of Job Number	27
4.4 Total Tasks Number	29
5 Conclusion and Future Work	30



List of Figures

1.1	Example of unnecessary speculative task	2
1.2	Example of special hardware request	4
2.1	HDFS architecture	6
2.2	Hadoop scheduling architecture	7
2.3	MapReduce WordCount example	7
2.4	MapReduce WordCount example with Combiner	8
2.5	Bad speculative task choice	9
2.6	Another example of unnecessary speculative task	11
2.7	Scheduling scheme for heterogeneous workload	12
3.1	Load-Aware scheduler architecture	15
3.2	SNMP query scheme	18
3.3	System loading versus Task execution time experiment	19
3.4	System loading versus Task execution time (a) node are not overloaded (b) node are overloaded	20
3.5	CPU frequency versus Task execution time experiment	21
4.1	Unnecessary speculative tasks avoided (a) total execution time (b) total execution time (c) number of speculative tasks	25
4.2	Dynamic loading detection (a) total execution time (b) number of speculative tasks	26
4.3	Impact of job number (a) total execution time (b) average tasktracker idle time (c) number of speculative tasks	28
4.4	Impact of total task number	29

Chapter 1

Introduction

Data are becoming larger and larger everyday and in every field. Google claims that in 2008, they obtain 20 petabytes of raw data everyday[1], while Facebook claims that they have processed over 12 terabytes of compressed raw data per day[2]. This phenomenon happened not only in web applications. For example, NYSE (New York Stock Exchange) generates 1 terabyte of data per day and LHC (Large Hadron Collider) in Geneva produce 15 petabytes of data every year[3]. These data are huge, hard to be stored, maintained, even processed. To address such problems, Google proposed a pack of distributed solution, including Google File System[4] which is a distributed file system that scales and automatically handles failure, MapReduce[1] which is a distributed programming framework that handles partial failure, recoverability, scalability and monitoring and BigTable[5] which is a distributed storage for structured data.

MapReduce framework have been widely discussed since [6] being published. Some researches leverage MapReduce to obtain scalability and performance enhancement for their applications. [7] discusses machine learning algorithms on MapReduce framework. [8] implements a co-clustering algorithm on MapReduce and points out that data mining algorithms need a good framework for large-scale data processing. Other papers like [9] and [10] see MapReduce as a good framework for data warehouse or storage base. There are also researches that implement MapReduce framework on special environments like GPU[11] or multi-core chips[12].

MapReduce is currently the most popular framework for large-scale data processing. MapReduce program adopts functional programming's paradigm so that it can be paralleled easily. Every MapReduce job has to implement two functions: map and reduce, according to the execution flow described in [1]. Given some input data, the framework will automatically partition

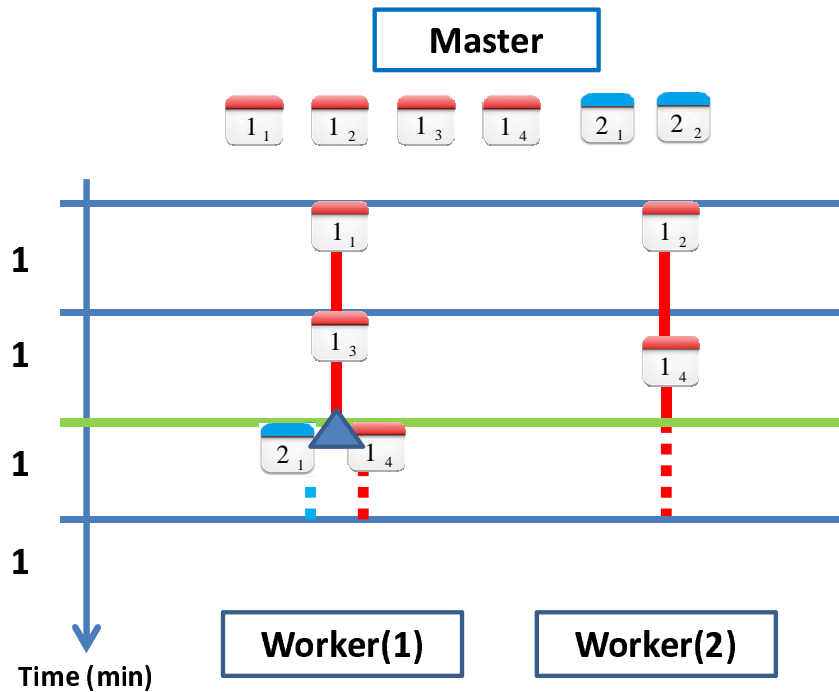


Figure 1.1: Example of unnecessary speculative task

the input data into several splits, and these splits are then processed by different instances of map function. Map function parses input data into key/value pairs, and for each pair, map function will output some or no intermediate key/value pair(s). All intermediate key/value pairs will be sorted so that values correspond to the same key will be aggregated together and processed by the same instance of reduce function. Each instance of reduce function will iterate over its value set, and output some or no key/value pair(s). Map function instances and reduce function instances are assigned to worker nodes by a master. If any node performs badly or crashes, the master will assign its current tasks to other worker nodes. This type of task is called "speculative task". It has been shown in [1] that using speculative task is able to reduce about 44% of response time.

In a practical cluster, there are two aspects to be concerned at MapReduce scheduling, "Job-wise" and "Task-wise". "Job-wise" scheduling decides which job or which job's task should be scheduled next, whether multiple jobs is able to run at the same time and how many resources a job or a groups of jobs can consume. "Job-wise" scheduling is necessary when jobs should be prioritized. Open source implementation Hadoop[13], which is the most popular open source implementation of MapReduce, provides a TaskScheduler interface for cluster administrators to design their own schedulers. Facebook and Yahoo! both contribute their schedulers named

FairScheduler[14] and CapacityScheduler[15] and the details will be given in Chapter 2. On the other hand, "Task-wise" scheduling decides which task to be ran and which worker node should the task be ran at. How tasks are being scheduled will greatly influence job's finish time. When some worker nodes perform poorly, speculative tasks can be launched to improve a job's response time. Previous study[16] has shown that Hadoop's heuristic of speculative tasks do not perform well in heterogeneous environment. As a result, the authors in [16] proposed a LATE scheduler to choose more suitable tasks to speculatively executed. The experimental result of [16] shows that job response time can improve up to factor of 2.

However, we observe that a MapReduce job might launch many unnecessary speculative tasks in the last wave of tasks. Consider the example in Figure 1.1. We have a MapReduce cluster with one master node and two worker nodes with different hardware. Worker one is 1.5 faster than worker two. A user submits two jobs, one with 4 tasks and the other with 2 tasks. Job one's task will run 1 minute on worker one and 1.5 minute on worker two. We can see in Figure 1.1, that at minute 2, we have reached a point to decide whether to launch a speculate task for job one, or to launch an ordinary task for job two. It is easy to tell that not to speculate the task is a better choice since launching a speculative task for job one will not make job one finish earlier. Our resources for such speculative task is totally wasted. Also, in this case, it is obvious to see that we could improve cluster utilization by scheduling job two's task to worker one at minute 2. If we could avoid unnecessary speculate tasks, we can finish more task in the same time. Moreover, some applications might leverage special hardware for optimization. For example, [17] points out that using CUDA GPU to do MRI reconstruction is 263 times faster than ordinary CPUs. We argue that MapReduce jobs should be able to specify their own job information, such as special hardware needed, special computing resources needed or rack restriction and master should schedule tasks according to the information it provides in order to gain better performance.

Default scheduler in Hadoop provides a FIFO queue for users. Hadoop MapReduce's implementation of master is called JobTracker and worker node is called TaskTracker. Jobs submitted by users are queued on JobTracker. JobTracker will not schedule any map tasks for the next job if all map tasks of the previous job are already scheduled and running. JobTracker will launch a speculative task if an active task's progress is a threshold behind the average task progress of that job. Hadoop MapReduce also does not provide any TaskTracker information to a job, an user cannot specify which TaskTracker its task should be ran at.

To summarize, we could improve the utilization of a MapReduce cluster by the following

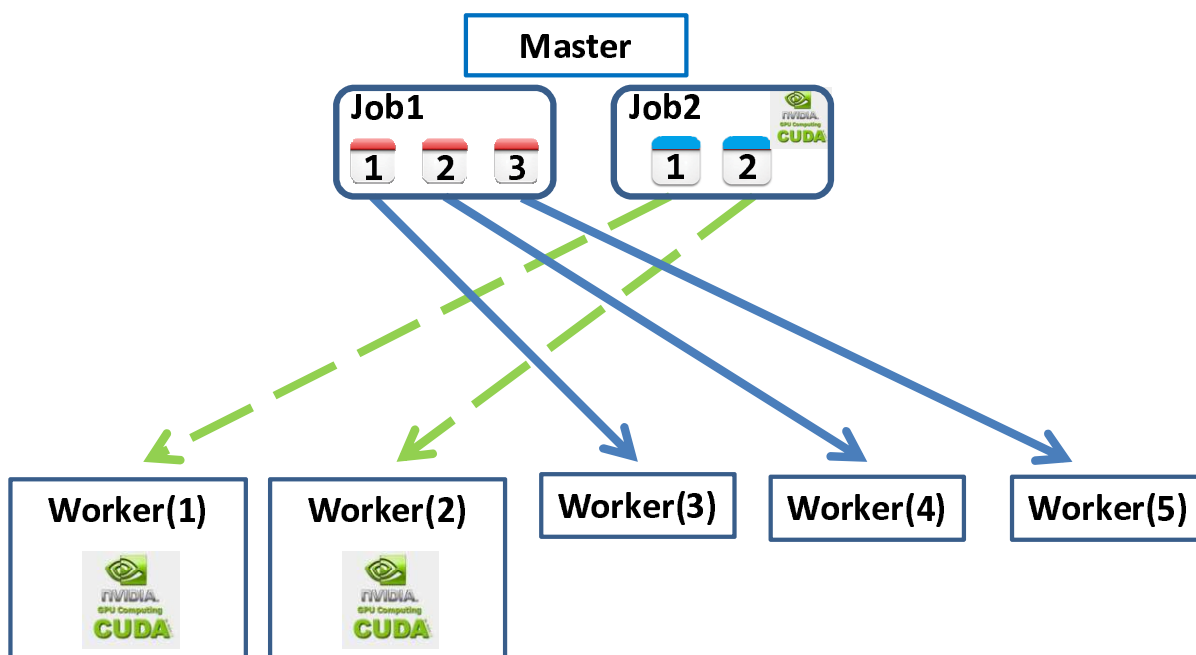


Figure 1.2: Example of special hardware request

two ways:

- Avoid unnecessary speculative tasks and schedule other active job's task instead.
- Schedule tasks according to its job's description, in order to assign tasks to suitable worker nodes.

We propose an add-on component that can be merged into MapReduce framework, and implement it on Hadoop MapReduce. The master in MapReduce framework is called JobTracker in Hadoop MapReduce. The worker node in MapReduce framework is called TaskTracker in Hadoop MapReduce. Map function instance in MapReduce framework is called mapper in Hadoop MapReduce. Reduce function instance in MapReduce framework is called reducer in Hadoop MapReduce. For further readability, above words are used interchangeably in following chapters.

This thesis is organized as follows: Chapter 2 will introduce some preliminaries for MapReduce Scheduling and related work. Chapter 3 gives a formal definition to our problem and describes our proposed solution. Experimental results are given in Chapter 4. Finally gives a conclusion in Chapter 5.

Chapter 2

Preliminary

2.1 Hadoop Architecture

Hadoop MapReduce is an open source implementation of MapReduce programming framework in the Apache Hadoop project. Hadoop MapReduce supports jobs to read input from many sources like HDFS, databases or ordinary file systems. Generally, MapReduce job tends to read its input from a distributed file system where MapReduce jobs can leverage data locality to process vast data. HDFS is an open-source implementation of GFS, and consists of one master NameNode and some slave DataNodes. When HDFS client needs to issue a read or write command, it first contacts the NameNode for meta information such as the block locations of its request file and the DataNodes where those blocks reside at. The client then issues the command to one DataNode holding the block, and transfers the block from the DataNode. Since data blocks are transferred directly from DataNode, master NameNode will not be a network bottleneck. HDFS has larger block size (64M by default) than ordinary file systems. Since the objective of HDFS is to read data sequentially from disk, using larger block size is able to reduce seek time resulting from data fragmentation. HDFS replicates data blocks to several DataNodes (3 by default), to use extra storage to trade for recoverability and data locality. Figure 2.1 illustrates a simple architecture of how HDFS client communicates with NameNode and DataNodes. Hadoop MapReduce consists of one JobTracker and some slave TaskTrackers. Users submit jobs to JobTracker, and every Hadoop MapReduce job contains a compiled jar file and a job configuration file. Job configuration can be specified by JobConf API or Configuration API in source code which the configuration file will be automatically generated. Users can also

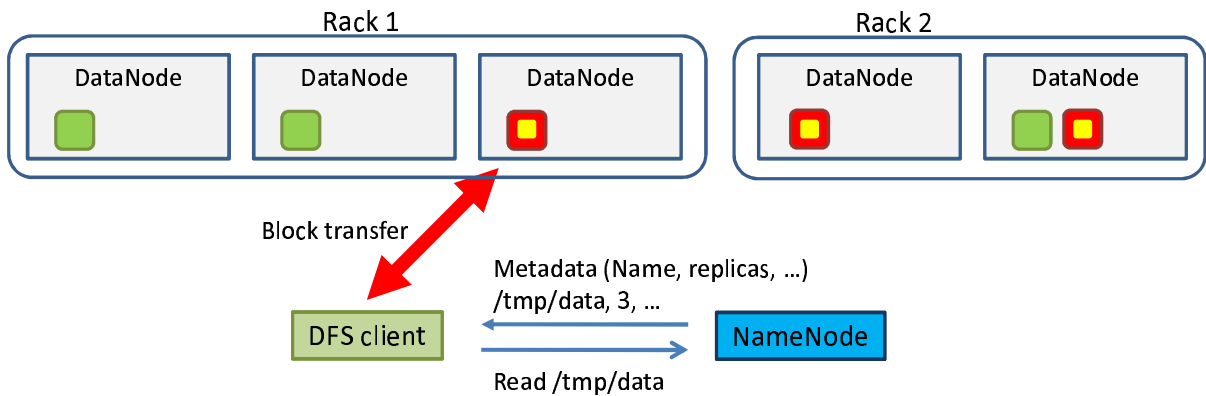


Figure 2.1: HDFS architecture

specify another XML file contains the configuration. JobTracker will parse the input path given in the configuration to several input splits based on the input block size and the number of map tasks specified in job configuration.

At map stage, we call every map instance a mapper, at reduce stage, we call every reduce instance a reducer. Every input split will be parsed by a class named RecordReader into many key/value pairs. Taking plain text file for example, each line will become a value, and its corresponding key is the offset of the original file. Mapper processes all the key/value pairs in its input splits and output some intermediate key/value pairs. These intermediate key/value pairs will be sorted, then all pairs with same key will be shipped to the same reducer as in Figure 2.3. The Combiner class can be used to reduce network communication if mappers tend to give out keys that appears more than once and local aggregation are feasible. Figure 2.4 shows how combiner can reduce the amount of data transfer on network after local aggregation. Reducers process input key/value pairs and output some or none key/value pairs. Users can also specify their own output format by overwriting RecordWriter class. Default RecordWriter write one key/value pair each line and key/value are tab-separated.

2.1.1 Hadoop MapReduce Scheduling

JobTracker is the core of Hadoop MapReduce scheduling. TaskTrackers will periodically send heartbeats to JobTracker. Heartbeat not only notifies the JobTracker that this TaskTracker is still alive, but also contains many important messages such as the number of empty working slots for mappers and for reducers that this TaskTracker has, or the progress of current active tasks this TaskTracker is processing. When a heartbeat is received, JobTracker will send back a heartbeat

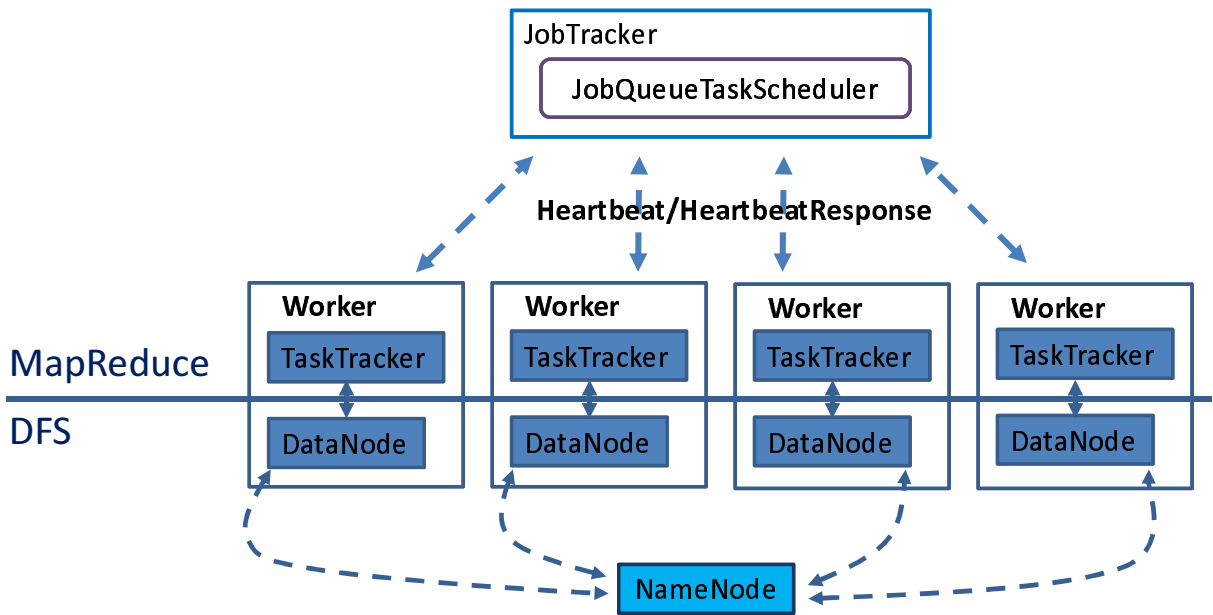


Figure 2.2: Hadoop scheduling architecture

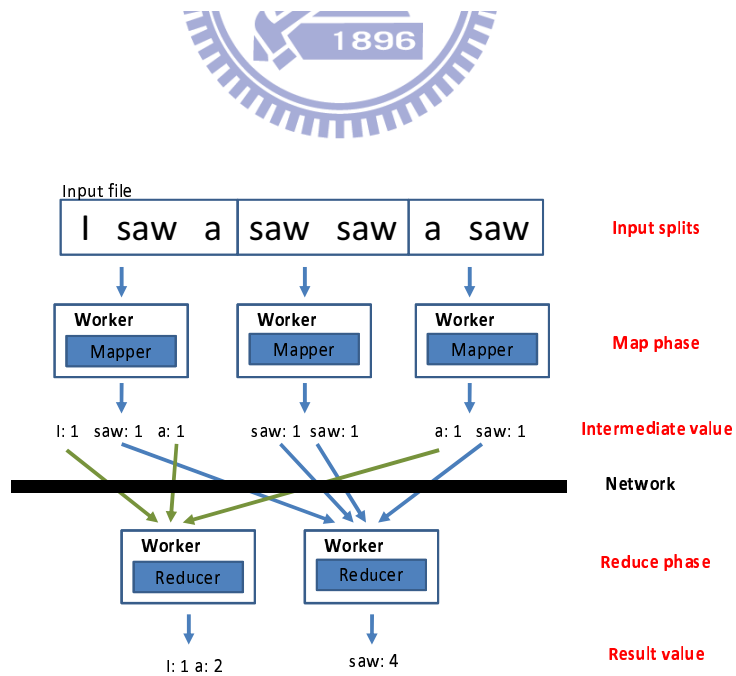


Figure 2.3: MapReduce WordCount example

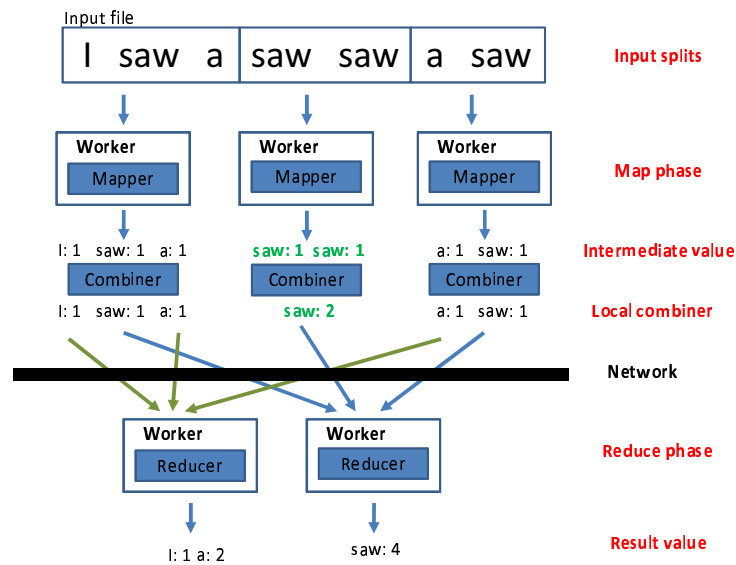


Figure 2.4: MapReduce WordCount example with Combiner

response. Hadoop MapReduce provides a TaskScheduler interface for administrators to specify how to assign tasks by overwriting some classes or methods. The default scheduler is called JobQueueTaskScheduler. All submitted jobs are in a FIFO queue and JobQueueTaskScheduler will later assign their tasks accordingly. All tasks that have failed before will be scheduled first. If there are tasks that reach the threshold of 0.2 below average progress, a speculative task will be scheduled.

2.1.2 Fair Scheduler and Capacity Scheduler

FairScheduler is an implementation of Hadoop MapReduce's TaskScheduler interface contributed by Facebook. FairScheduler categorizes jobs into pools, and shares resources fairly among these pools. By default, each user owns a pool when his/her job is submitted to JobTracker. Administrators can also redefine pools to aggregate some groups of users' jobs into the same pool. Pools can be dynamically configured, and FairScheduler will reload pool configuration every 15 seconds. Every job is considered active when submitted. Administrators can specify maximum running jobs per pool, and all jobs in the same pool will share the resources fairly. Pools can also be defined with priorities, and every pool can have a weight value showing the percentage of resources should FairScheduler be given to. Whenever some resources are unused, excess resources are evenly split between pools. FairScheduler provides three extensible classes, WeightAdjuster, LoadManager and TaskSelector. WeightAdjuster is used to adjust the

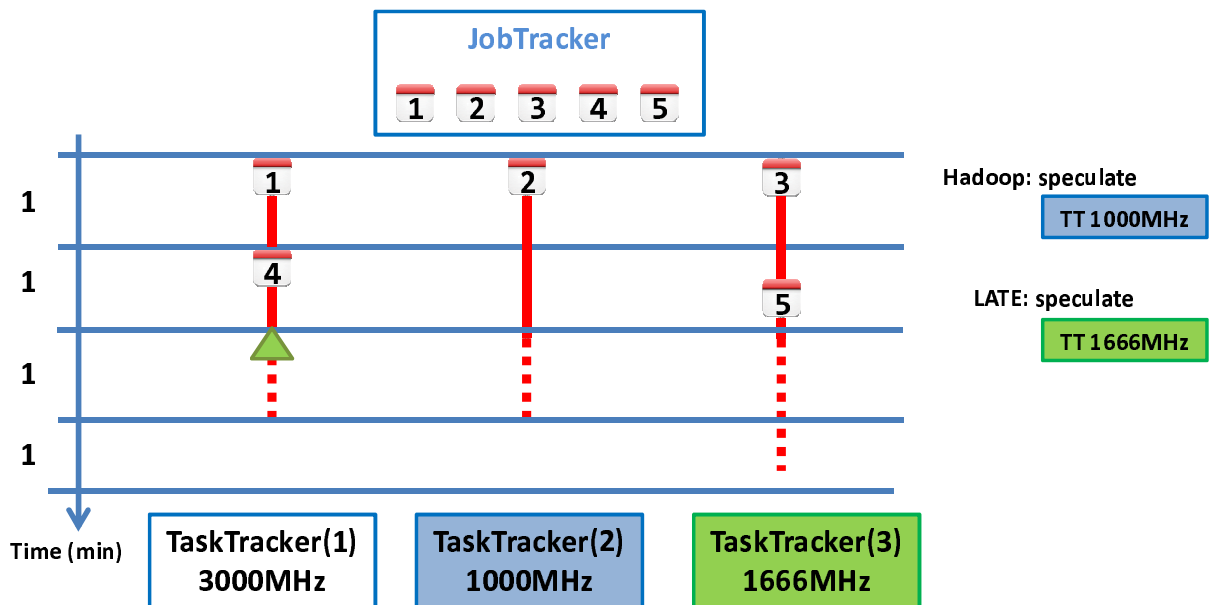


Figure 2.5: Bad speculative task choice

weight of running jobs. Given some weight boost to new coming job is proven to be effective for cluster running a lot of small jobs[18]. LoadManager is used to dynamically determine how many maps and reduces can run on a given TaskTracker. TaskSelector is used to determine if some tasks can run on a specific TaskTracker.

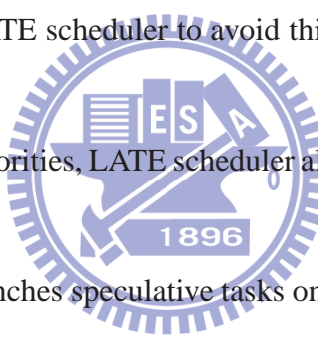
CapacityScheduler is another implementation of Hadoop MapReduce's TaskScheduler interface contributed by Yahoo!. CapacityScheduler categorizes jobs into queues. All queues must be predefined with a percentage as their share of the whole resource capacity. All jobs in the same queue are in a FIFO order, and only one job is active at any time. Like FairScheduler, free resources can be given to queues beyond its capacity to improve utilization. CapacityScheduler supports memory intensive jobs. A job can specify memory requirements, and the tasks of the job will only run on TaskTrackers that can meet the memory requirement of the job. Memory based scheduling is currently only supported in Linux platform.

2.2 Related Work

2.2.1 LATE Scheduler

The importance of speculative execution in MapReduce framework is discussed and improved in [16]. [16] points out that the original Hadoop heuristic toward speculative execution be-

comes a disappointment in heterogeneous environments. Default Hadoop scheduler assumes that nodes are homogeneous and tasks tend to finish in waves. Heuristic built on this assumption will result in some bad choices in choosing which tasks to be speculatively executed. We use the following example to illustrate how the default scheduler speculates the wrong task. Suppose we have a MapReduce job with 5 tasks and a cluster with 3 TaskTrackers. Each TaskTracker has different process rates: 3000MHz, 1000MHz and 1666MHz as in Figure 2.5. At first, task 1, 2 and 3 are assigned to 3 TaskTrackers and estimated to finish in 1, 3 and 1.8 minutes. After one minute, TaskTracker one finishes its task and task 4 is assigned by JobTracker. 0.8 minute later, TaskTracker three finishes its task and task 5 is assigned by JobTracker. 0.2 minute later, TaskTracker one finishes its task, so it has an empty working slot. Task 2 and task 5 are both qualified to launch speculative tasks. It is obvious that launching a speculative task for task 5 can help the job finish within 3 minutes while launching speculative task for task 2 will not help at all. In above case, default scheduler might speculatively executed task 2 at minute 2. [16] designed the LATE scheduler to avoid this sort of speculation by the following three principles:

- 
- Given task speculative priorities, LATE scheduler always speculates the task of the longest estimated time to finish.
 - LATE scheduler only launches speculative tasks on fast nodes.
 - Limit the number of concurrent speculative tasks to prevent thrashing.

LATE scheduler can significantly improve a single job's response time. However, LATE did not consider the fact that there might be some unnecessary speculative tasks. Consider the example in Figure 2.6, TaskTracker one will not be helpful neither speculating task 2 nor task 5. LATE scheduler also did not consider the fact that node performance might vary from time to time. LATE scheduler uses a heuristic to evaluate node performance by the sum of its previous working progress. This heuristic might misjudge some nodes as slow nodes if those nodes happened to have some short, non-MapReduce, high-load process running.

2.2.2 Heterogeneous Workloads

[19] tried to improve hardware utilization by distinguishing different kinds of workloads of MapReduce jobs. When CPU-bound and I/O-bound jobs run in parallel, both CPU and I/O

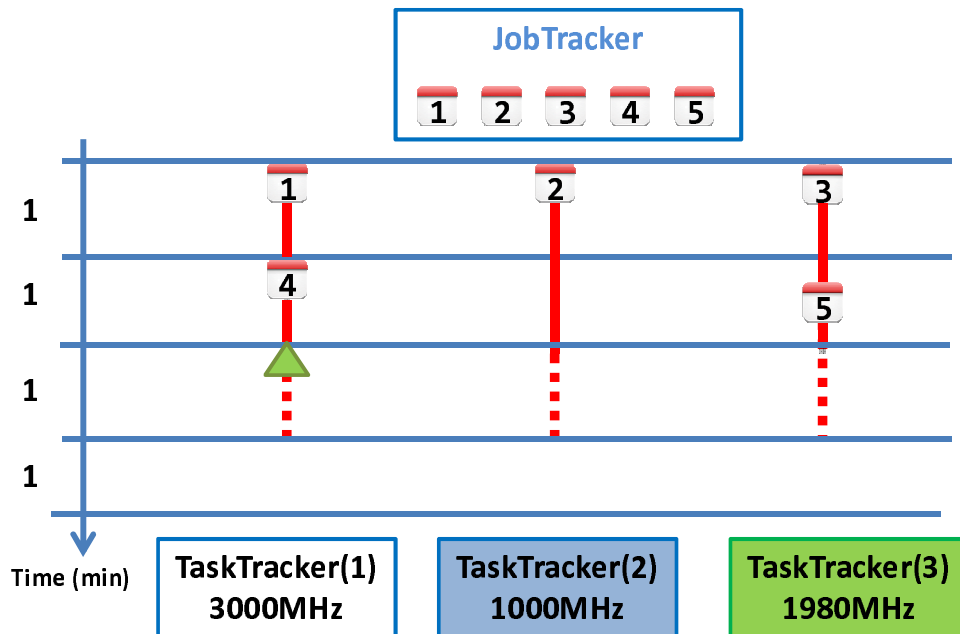


Figure 2.6: Another example of unnecessary speculative task

devices can contribute to the jobs. [19] proposed an MR-Predict mechanism to predict whether a job is CPU-bound or I/O-bound. Figure 2.7 shows the architecture of [19]. [19] designed a triple-queue scheduler that maintains three queues: wait queue, CPU-bound queue and I/O-bound queue. When a new job being submitted, the scheduler put the job into the wait queue. For jobs in the wait queue, the scheduler will schedule one of its task. After the task finishes, the MR-Predict mechanism will predict whether the job is CPU-bound or I/O-bound based on its running status. For jobs predict as CPU-bound, it will be put into the CPU-bound queue. For jobs predict as I/O-bound, it will be put into the I/O-bound queue. Both CPU-bound queue and I/O-bound queue are FCFS. Jobs in CPU-bound queue and I/O-bound queues run parallelly, thus improving hardware utilization.

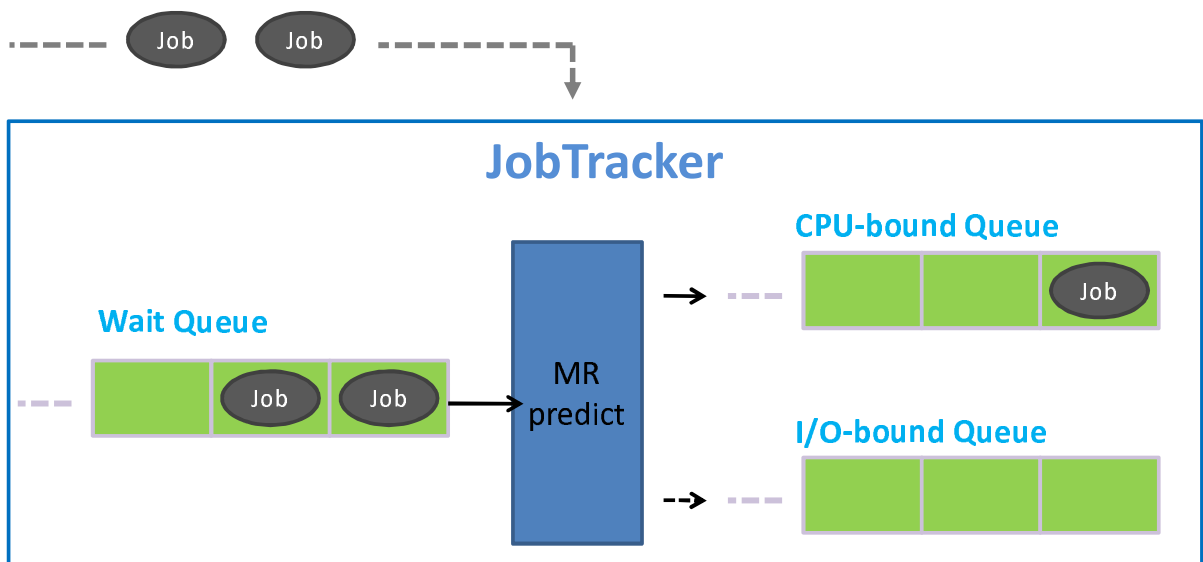
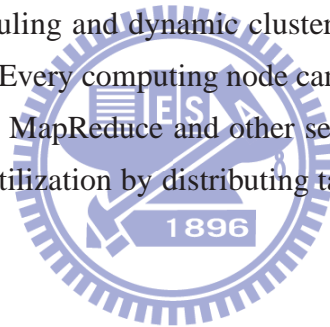


Figure 2.7: Scheduling scheme for heterogeneous workload

Chapter 3

Load-Aware Scheduler

This chapter describes our optimization target and cluster environment. Our goal is to improve the overall utilization of a MapReduce cluster. Default Hadoop MapReduce scheduler and [16] do not take multiple job scheduling and dynamic cluster status into account. Our computing environment is heterogeneous. Every computing node can have different computing capability. Every computing node can run MapReduce and other services simultaneously. Our proposed scheduler will try to improve utilization by distributing tasks according to the working nodes' status at any given time.



3.1 Objectives

We further list three objectives of our scheduler:

- For tasks whom might be considered to be speculatively executed, evaluate whether launching this speculative task will shorten its job finish time. If the original task will finish before the speculative task or other ongoing tasks of this job will not finish after this speculative task. We should not speculatively execute the task and leave the available resources to other active jobs.
- Choose fast nodes to speculate tasks. We should acquire more information about a TaskTracker when we judge how this node will perform. There might be other processes on the same node compete for the resources with MapReduce tasks. It is important to have an interface to fetch the information on a given TaskTracker. With that information, we

can further tell if this TaskTracker could outperform itself at a given past time t or if it could outperform another TaskTracker.

- Information from MapReduce jobs is as important as information about a TaskTracker. Users may list the requirements that will benefit the job, and JobTracker should try to meet these requirements in order to make the job run smoother.

3.2 Proposed Scheme

To achieve the above objectives, we propose our Load-Aware scheduler. It consists of two components: Data collection module and task scheduling module. Our scheduler follows the following two guidelines:

- When a TaskTracker has an empty working slot, assign a task to the TaskTracker only when assigning this task could make its job finish earlier.
- For any active job, only TaskTrackers that are not overloaded and can fulfill the job's requirements (if any) can be assigned the job's task to.

Our scheduling idea works simply as follow: For running jobs and tasks, our scheduler will use the information collected by data collection module to compute their estimated finish time. When a TaskTracker is available and asks for tasks, our scheduler iterates over the active job list. For each job, we applied the following steps:

- Step 1: Check if the TaskTracker can fulfill the requirements listed in job's configuration. If the TaskTracker cannot, continue to next job.
- Step 2: Check if there are tasks of the job need to be scheduled (including fail tasks, non-running tasks and tasks need to speculate). If there is none, continue to next job.
- Step 3: Estimate the task running time on the TaskTracker, check if schedule this task can shorten the job's finish time. If it cannot, continue to next job.
- Step 4: Assign a task of current job to the TaskTracker.

Figure 3.1 shows our proposed architecture. We implement our two modules on JobTracker. Data collection module communicates with TaskTrackers and provides information for task scheduling module. Task scheduling module makes scheduling decisions based on information from JobTracker and data collection module.

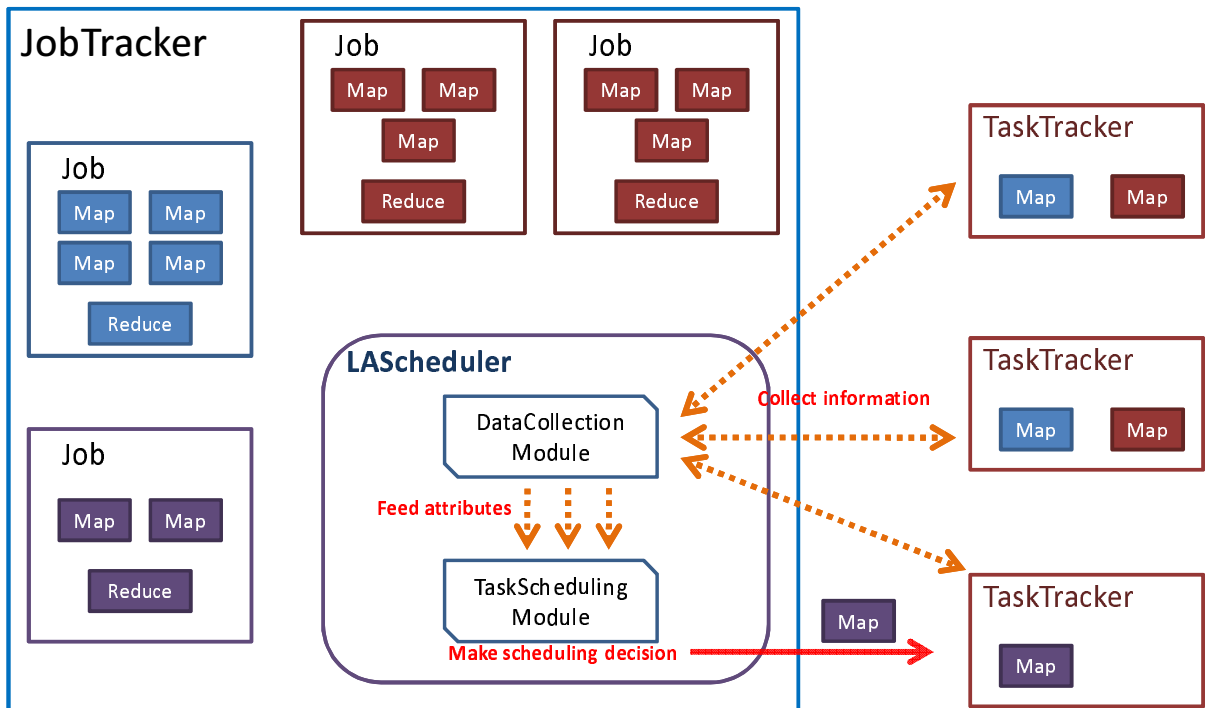


Figure 3.1: Load-Aware scheduler architecture

3.2.1 Data Collection Module

Data collection module provides an interface for JobTracker to gather necessary information of TaskTracker for later scheduling decisions. We need information about a TaskTracker to see if it can meet the requirements of a job. We also need information about TaskTrackers to evaluate the running time of a task. Any attribute that might affect the performance of a TaskTracker can be put into the collection list on data collection module. CPU frequency, system loading, I/O rate and memory usage are some general attributes that might be selected as our evaluation metrics. Other than that, CPU core number, special hardware information and network usage can be useful from time to time. Other miscellaneous information such as task progress, task status, task starting time and task finishing time are also needed to be collected by the module. Data collection module mainly works on JobTracker, and data collection should run periodically to collect information about all TaskTrackers. Further implementation details will be given in Section 3.3.

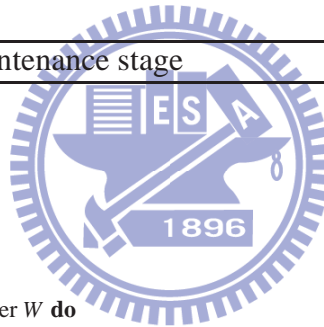
3.2.2 Task Scheduling Module

While data collection module gathers information, task scheduling module maintains information and makes scheduling decisions when empty working slots are available. There are two stages in task scheduling module: information maintenance stage and task assignment stage. Information maintenance stage is responsible for the estimation of running task execution time. We need to give an estimation of running tasks in order to estimate the finish time of the whole job. With the estimation, we can further decide whether scheduling another task could not be a waste. Finished tasks and running tasks can be an important reference for future estimation. We need to keep track of the dynamic attributes to make more accurate estimation. Taking system loading for example, system loading could be an important factor to the performance of a TaskTracker and it changes dynamically. TaskTracker might performs differently when other processes on the same node compete for its resources. System loading can help us distinguish the status of a TaskTracker.

Algorithm 1 : Information maintenance stage

Input: M : JobTracker of MapReduce cluster Q : active job queue on JobTracker M **Periodically:**

- 1: **for** each job j , j in Q on M **do**
 - 2: **for** each task t , t in j on TaskTracker W **do**
 - 3: Update progress for t .
 - 4: Calculate estimated finish time for t .
 - 4: Update and record dynamic attributes status for t on W .
 - 5: **end for**
 - 6: Calculate estimated finish time for j .
 - 7: **end for**
-



Whenever an empty working slot is available on a TaskTracker, JobTracker will iterate over active jobs. For each job, JobTracker needs to evaluate the task execution time on the TaskTracker if the TaskTracker meets the jobs' requirements. We should iterate over all attributes that affect the performance of a TaskTracker, and adjust the finish time according to the difference between past and current situations. Note that to complete a MapReduce job is to complete a series of tasks of that job. When a job is about to finish, some of its tasks must have finished before. So there are always some task finish time references for us. In order to make adjust-

ments for each attribute, some adjustment functions are needed. We use CPU-intensive jobs as our experiment applications. In Section 4, we will demonstrate how to estimate task finish time by observing system loading and CPU frequency.

Algorithm 2 : Task assignment stage

Input:

M : JobTracker of MapReduce cluster

Q : active job queue on JobTracker M

R : requirements list of a job

L : list of attributes

Whenever a TaskTracker N have an empty working slot:

```
1: Task assignTask( TaskTracker  $N$ ) do
2:   for each job  $j$ ,  $j$  in  $Q$  do
3:     for each requirement  $r$ ,  $r$  in  $R$  do
4:       if (  $N$  cannot meet  $r$  ) then
5:         Next job.
6:       end if
7:     end for
8:     for each attributes  $a$ ,  $a$  in  $L$  do
9:       Estimate task execute time  $T$  on  $N$  over  $a$ .
10:    end for
11:    if (  $T$  > estimate finish time of  $j$  ) then
12:      Next job.
13:    else
14:      Assign task of  $j$  to  $N$ .
15:    end if
16:  end for
17: end assignTask
```



3.3 Implementation

We choose Hadoop MapReduce as our implementation platform. To make our scheduler more flexible and extensible, we merge our two modules into FairScheduler, which is pluggable from Hadoop.

We modified FairScheduler to achieve above objectives. Our scheduler still retains the fairness characteristic of FairScheduler since we only avoid unnecessary speculative tasks. We schedule tasks based on the configuration of the job itself.

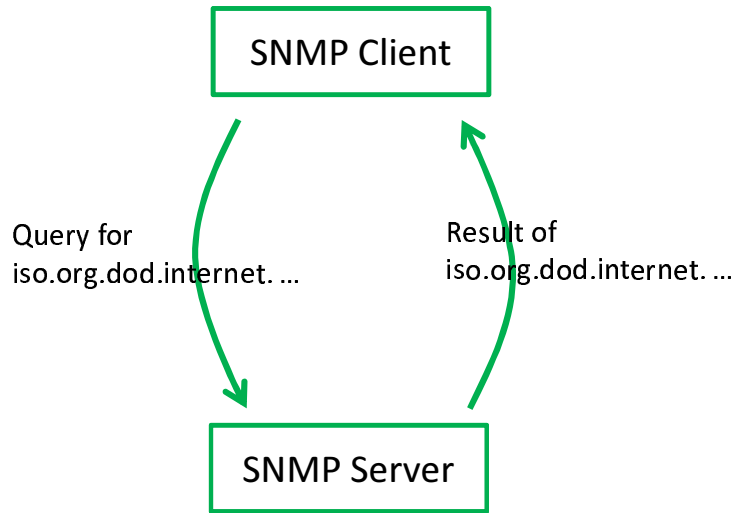


Figure 3.2: SNMP query scheme

For data collection module, we need an information exchange approach between JobTracker and TaskTracker. We can design our own protocol to exchange information. This approach might lead the scheduler to inflexible and hard to implement. Taking an attribute we might want to collect for example. We want to obtain system loading on a TaskTracker. Different operating systems might need different system calls or different programs to obtain. For each operating system we want to support, we will need a set of codes dealing with local data extraction in TaskTracker. Not only did we need the effort to write extra codes, if there are any new platforms we want to support, we have to re-compile the source codes and restart the service. To avoid these disadvantages, we choose SNMP[20] as our data collection protocol. SNMP (Simple Network Management Protocol) is a protocol designed for network management and network monitoring. SNMP defines a structure of management information and the management information base[21]. Figure 3.2 shows an example of a SNMP query flow. We choose Net-SNMP[22] as our SNMP server on TaskTracker. Net-SNMP is originally developed by University of California, Davis and is now an open-source project. Data collection module in JobTracker will send SNMP query to TaskTracker periodically. There is much information already defined in management information base (MIB). Taking previous system loading for example, query iso.org.dod.internet.private.enterprise.ucdavis.laTable.laEntry.laLoad (MIB number 1.3.6.1.4.1.2021.10.1.3) can obtain the system loading of a TaskTracker when using Net-SNMP. Net-SNMP defines a extTable (MIB number 1.3.6.1.4.1.2021.8) for administrators to specify self-defined commands. Given a program, Net-SNMP will run the program and re-

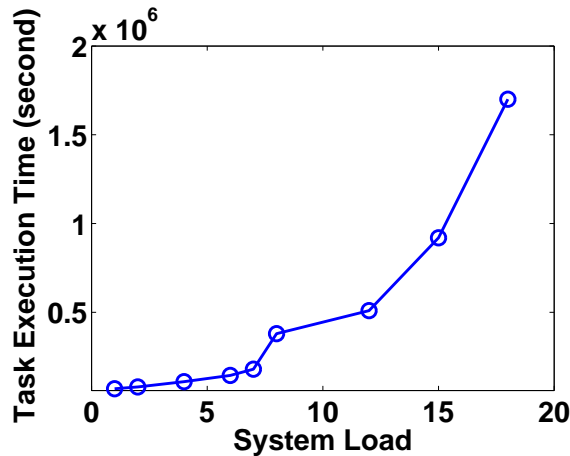


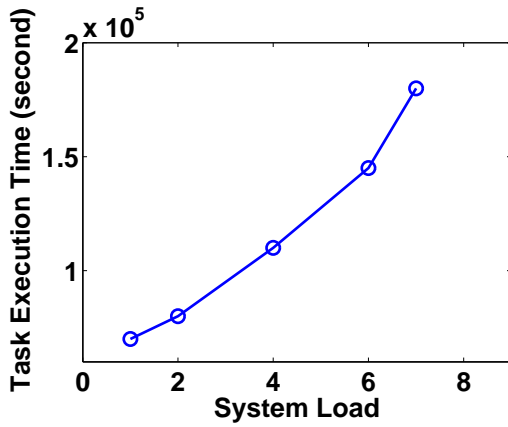
Figure 3.3: System loading versus Task execution time experiment

turn its output as the result of an MIB entry. We leverage the `extTable` to construct our structure to do jobs' requirement checking. MapReduce jobs can set a special flag expressing that they need some checking for TaskTracker. Only if all the results of SNMP query fit the job's requirements will the JobTracker assign tasks to the TaskTracker. To add a new checking attribute for MapReduce jobs, we applied following three steps:

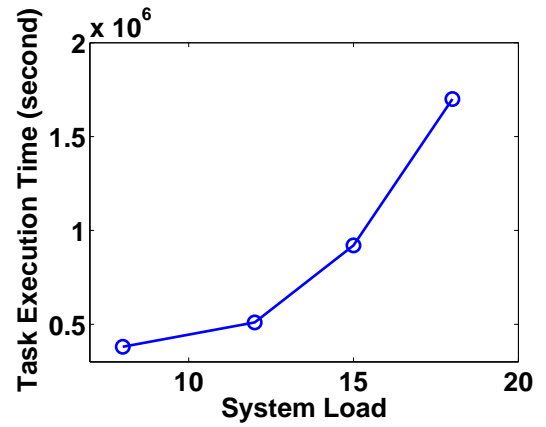
- Add the corresponding program on TaskTracker.
- Restart the SNMP daemon on TaskTracker.
- Notify users the MIB number of the attribute.

We do not have to restart our MapReduce service thus no active jobs and users are influence by this process.

For task scheduling module, we leave the speculation threshold unchanged and focus on avoiding unnecessary speculative tasks. Our work needs an estimation approach toward task execution time. Our target application is CPU-intensive. We analyze two attributes that we think have the most effect on task execution time: system loading and CPU frequency. System loading gives a global view of CPU usage about a TaskTracker. High loading means fewer resources can be obtained by one process. We need an estimation function that gives task estimated execution time based on previous task executing experience. Given current loading and previous loading plus running time, we can predict how long the task might elapse on this TaskTracker. We perform an experiment to observe the relation between loading and task execution time. The



(a)



(b)

Figure 3.4: System loading versus Task execution time (a) node are not overloaded (b) node are overloaded

results are shown in Figure 3.3. Multi-core computers are common nowadays. We found that the growing curve differs when system loading grows larger than system core number.

Our experiment node has 8 cores. We can see in Figure 3.3 that when x-axis reached beyond 8, the curve grows exponentially. When system loading lower than 8, the curve linearly grows. We split the experimental result in half, and analyze the changes before and after system loading reaches core number in Figure 3.3.

Base on the experiment, we have two estimation functions of system loading:

- $Time' = 17700 * (Load' - Load) * Time$ when node are not overloaded.
- $Time' = Time * e^{0.12 * (Load' - Load)}$ when node are overloaded.

We use the same technique for CPU frequency. We perform an experiment to observe the relation between CPU frequency and task execution time. The results are shown in Figure 3.5. Base on the experiment, we have an estimation function of CPU frequency:

- $Time' = Time * Freq' / Freq$

When task scheduling module needs an estimation of some task, following steps are taken:

- Select an executed task of the same job.
- Estimate the finish time based on system loading adjustment function if selected task is processed by the same TaskTracker.
- Estimate the finish time based on CPU Frequency and system loading adjustment functions if selected task is not processed by the same TaskTracker.

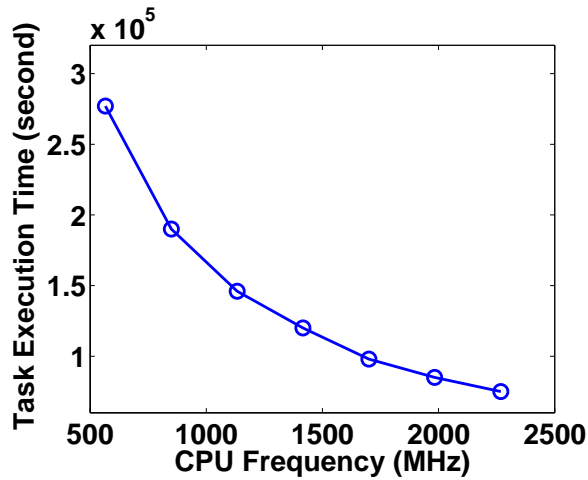


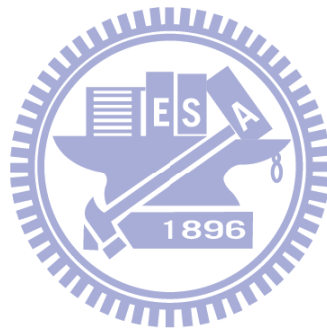
Figure 3.5: CPU frequency versus Task execution time experiment

3.4 Discussion

While Hadoop MapReduce was designed to use on large-scale cluster, its elegant framework design has made many small organizations push their applications on top of it. Smaller scale clusters are common in campus to support courses and researches. Companies that are not web-based can also store and analyze data on top of Hadoop MapReduce. Heartbeat plays an important role on the architecture of Hadoop MapReduce. JobTracker and TaskTracker mainly communicate over heartbeats and its responses. JobTracker needs to process heartbeats and gives proper responses. To avoid overwhelming the JobTracker, current Hadoop MapReduce has a 3 seconds lower-bound for heartbeat interval. Heartbeat interval grows linearly when cluster size increases. In our experience, a 3 seconds lower-bound for heartbeat interval seems quite much for small cluster. Jobs submitted in cluster cannot be scheduled until next heartbeat was sent by a TaskTracker who has available working slots. Small jobs that take less than 1 minute might waste over 10% of time waiting for heartbeats. There are some discussions about dynamic adjustment of heartbeat interval on Hadoop MapReduce. Issue number 5784[23] discusses about configurable heartbeat interval. [24] lowers the minimum heartbeat threshold on small cluster. The experiment of [24] receives quite a stunning performance burst of 100% on a small cluster.

Heartbeat between JobTracker and TaskTracker carries much information around. If the information that data collection module needed can be padded on heartbeat, it will be undoubtedly be an enhancement of performance. There are issues [25] [26] that discussed about padding ex-

tra information on heartbeats. In the future, our data collection module can leverage the extra information already carried by heartbeat.



Chapter 4

Performance Evaluation

In this chapter, we evaluate our scheduler in a real-world environment. We create some scenarios including multiple jobs and dynamic loading. We extend a local sudoku game solver to a MapReduce version that solves multiple sudoku games as our test application. Input sudoku games are randomly generated in advanced by our soduku game generator.

Our testbed is a local cluster with 8 nodes. Each node has 1 Xeon CPU with 4 cores supporting Hyper-Threading, 12GB of RAM and a 500GB SATA drive. All nodes are connected on a gigabit Ethernet channel. Solving 9000 games on one of our machines locally takes about 500 seconds. Solving 9000 games on 8 nodes using Hadoop MapReduce takes only 80 seconds, each mapper takes about 60 seconds.

Our experiment scenario consists of a set of jobs and the status of each TaskTracker. Status of a TaskTracker might vary from time to time in order to observe the effect of dynamic loading. Two performance metrics are used in our experimental result, the total execution time of a set of jobs and the number of speculative tasks these jobs launched.

In the following sections, we refer our scheduler as LA, scheduler of [16] as LATE and default Hadoop MapReduce scheduler in version 0.20.2 as default.

4.1 Unnecessary Speculative Tasks

Speculative tasks always launch at the last wave of tasks. To speculatively execute a task means a part of the resources of this cluster is unable serve other jobs. We want to observe the effect of unnecessary speculative tasks hogging resources, which cause performance degradation.

We create heterogeneous environments by two ways: increase system loading and adjust CPU frequency on TaskTracker.

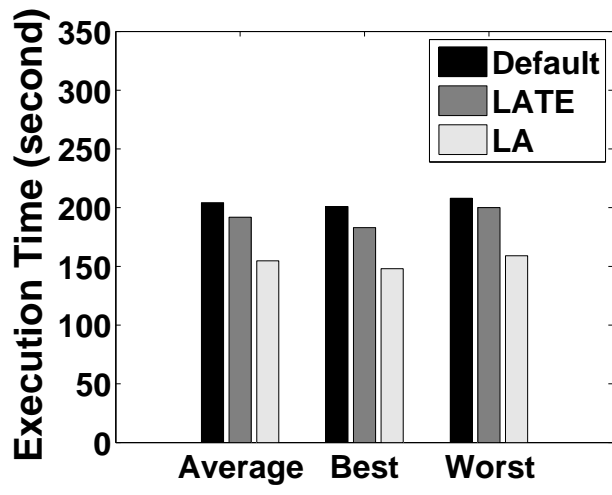
We increase system loading by running several other CPU-bound processes on one of our TaskTrackers. TaskTracker's loading will remain 6 when processing new coming tasks. We submit two jobs into our cluster. Each job takes 9000 sudoku games as their input. When the first job is close to finish, some speculative tasks might be launched when LATE or default Hadoop scheduler are applied. Figure 4.1 (a) shows the total execution time of this experiment. LA outperforms the other schedulers 25% on average.

With the same job setting, we adjust one of our TaskTracker's CPU Frequency from 2268MHz to 1416MHz but without the extra loading. Figure 4.1 (b) shows the total execution time of this experiment. LA outperforms the other schedulers 20% on average.

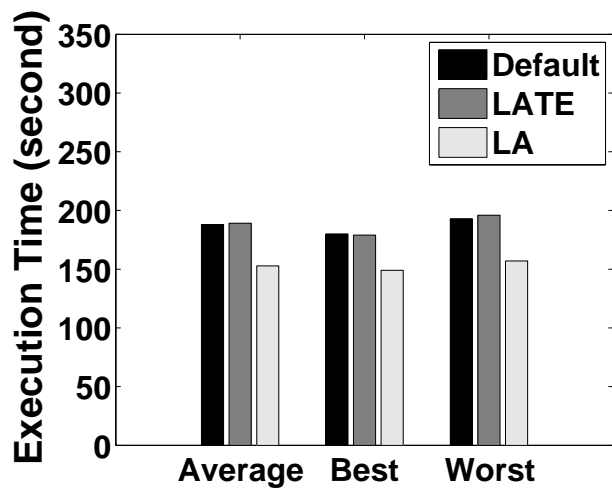
Without checking whether the speculative tasks are necessary, TaskTrackers waste their CPU time on speculative tasks that never finish before original tasks. LATE scheduler chooses the better task to speculatively executed. Speculative tasks are prioritized but the number of speculative tasks is not decreased. All tasks that meet the threshold of speculation are speculatively executed if there are enough free resources. LA evaluates which speculative task might finish after the original task and not to launch these tasks. Figure 4.1 (c) shows the number of speculative tasks launched for each scheduler in this experiment. LA launches less speculative tasks and re-schedules these resources to the second job.

4.2 Dynamic Loading

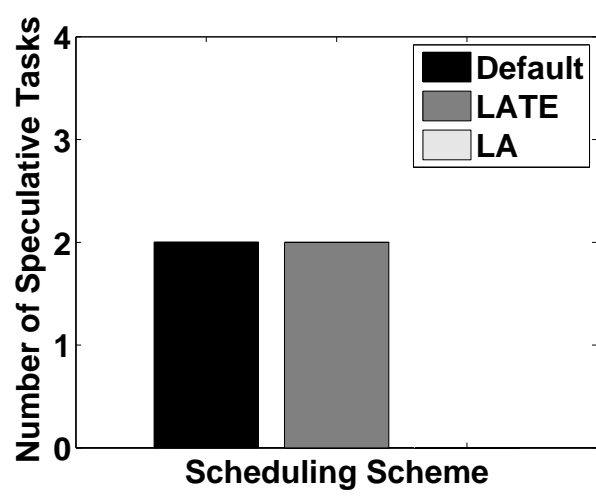
Dynamic loading is common when the same group of hardware runs multiple services. Equal hardware capability does not mean equal performance. One machine can be busy at this minute, but load-free at the next minute. We use system loading as a reference of current TaskTracker's status. By keeping track of the system loading of TaskTrackers, we can estimate whether the same task will run faster or slower. We create a dynamic loading scenario as follow. We submit 4 jobs into our cluster. The first job processes 30000 sudoku games. The second job processes 20000 sudoku games. The third job processes 10000 sudoku games. The last job processes 10000 sudoku games. 4 out of 8 TaskTrackers are swamped by other processes with loading 4 at the beginning. 2 out of 4 TaskTrackers are given extra load to loading 8 after three minutes; the other 2 TaskTrackers are freed from loading.



(a)

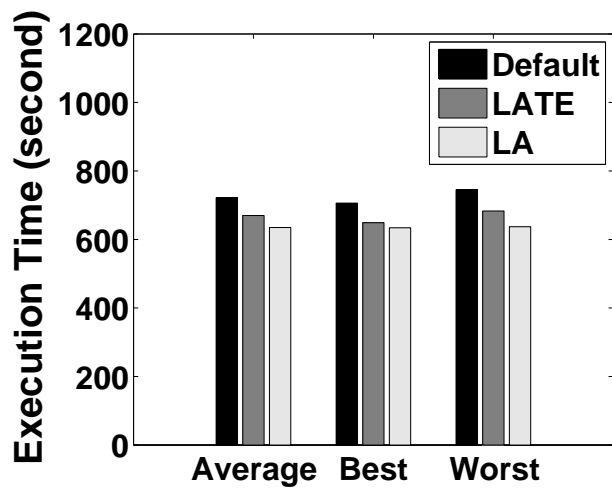


(b)

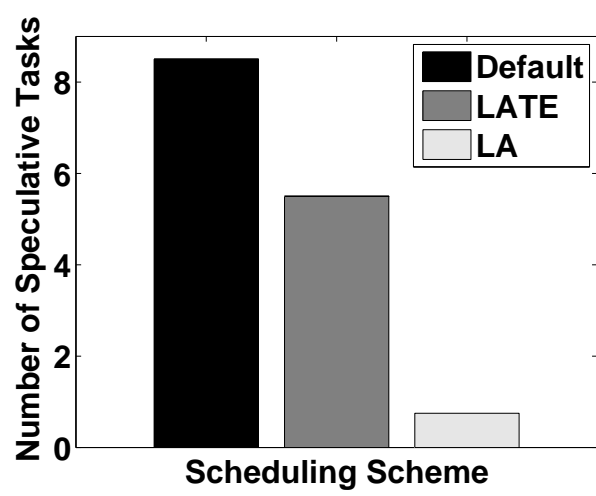


(c)

Figure 4.1: Unnecessary speculative tasks avoided (a) total execution time (b) total execution time (c) number of speculative tasks



(a)



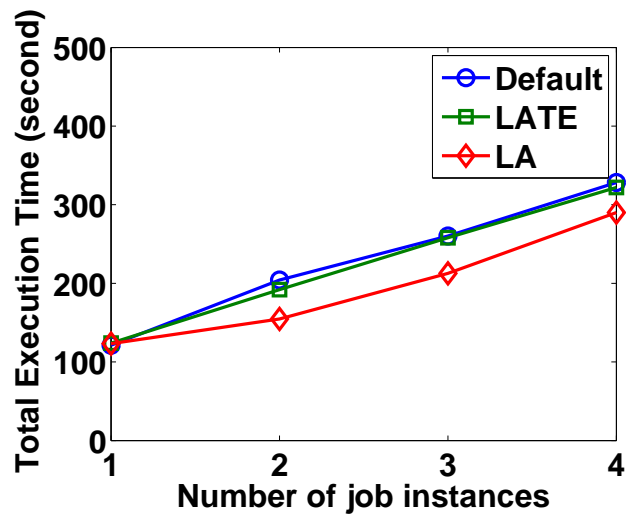
(b)

Figure 4.2: Dynamic loading detection (a) total execution time (b) number of speculative tasks

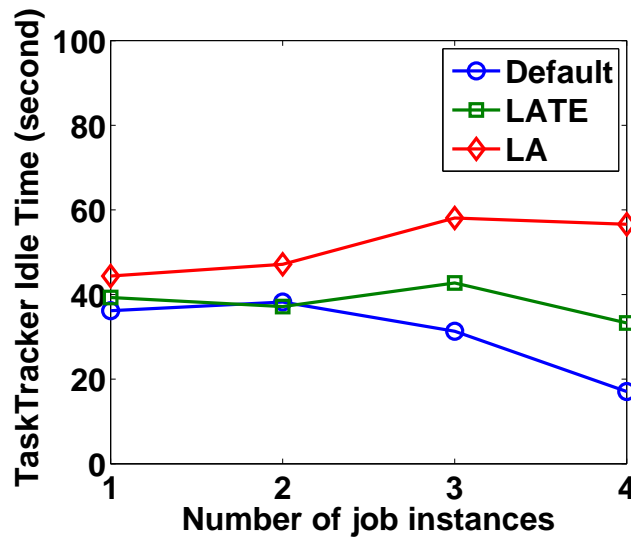
Default Hadoop scheduler pays no attention to a node's status. If an active task performs badly, a speculative task will be launched. LATE scheduler records the history of how the node performed, and launches speculative tasks only on nodes which performs better in the past. 2 out of 4 TaskTrackers are load-free after three minutes, and can be scheduled some speculative task to help finish second or third job faster. LATE will still see these nodes as bad performer. LATE stops scheduling any speculative tasks on these nodes. LATE performs better than original Hadoop scheduler since it has a SpeculativeCap. Maximum speculative tasks can be launched at the same time are limited by this parameter. In our experiment, we set SpeculativeCap to 2. Because of SpeculativeCap, LATE avoids some unnecessary speculative tasks. LA detects system loading on TaskTrackers. When TaskTrackers ask for tasks, LA fetches its current loading and estimates task finish time. If the speculative task can shorten the finish time of the job, LA will schedule it. Figure 4.2 (a) shows the total execution time of this dynamic loading scenario. We can see that LA can improve up to 15% of execution time. Figure 4.2 (b) shows that we avoid launching 90% of speculative tasks.

4.3 Impact of Job Number

[1] and [16] shown that launching speculative tasks can improve job response time. LA stops speculatively execute some tasks at some point. We need to know that tasks that need to be speculatively executed are launched so that job's response time is not hurt. We are also curious about whether our scheduler could achieve same improvement when user submit more jobs. We vary the number of jobs submitted into our cluster. Each job takes 9000 sudoku games. We increase system loading of one of our TaskTracker to 6. The results are shown in Figure 4.3. We can see that when there is only one job, LA performs just like default scheduler and LATE. Figure 4.3 (c) shows that we avoid more speculative tasks when number of jobs increase. Figure 4.3 (a) shows that while number of jobs increase, the difference of total execution time between LA and other two schedulers are almost the same. Meaning that the improvement percentage become lower when number of jobs increase. Total execution time of a set of MapReduce jobs is determined by the last finish task of last job. If we are lucky enough that the working slots for the unnecessary speculative tasks we avoided are filled and finish some other active jobs, we could have more improvement on total execution time. On the other hand, LA could seem no improvement on total execution time like the case where there is only one MapReduce job.



(a)



(b)



(c)

Figure 4.3: Impact of job number (a) total execution time (b) average tasktracker idle time (c) number of speculative tasks

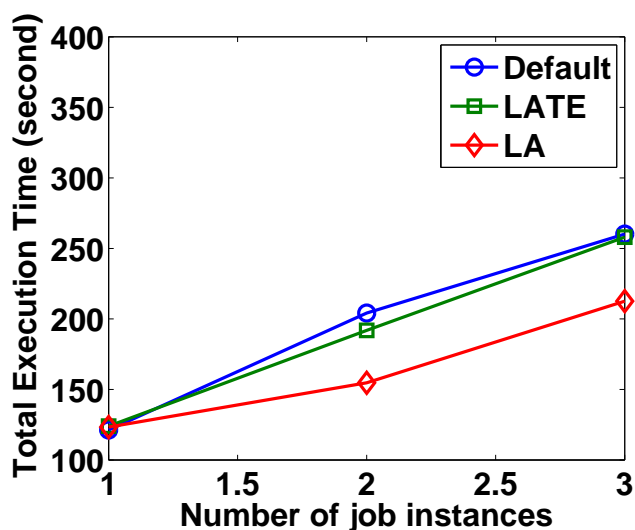


Figure 4.4: Impact of total task number

By avoiding unnecessary speculative tasks, TaskTracker becomes idle and ready to serve other tasks. We observe the idle time of our TaskTrackers in the same experiment. The results are shown in Figure 4.3 (b). We can see that the difference between speculative tasks launched is directly related to the difference between TaskTracker idle time. The more unnecessary speculative tasks we avoid, the more time TaskTrackers idles. If there are other tasks we could schedule while TaskTrackers idles, the overall utilization becomes higher.

4.4 Total Tasks Number

Speculative tasks only launched at last wave. There are less rooms for us to improve if the job is disassembled into more tasks. LA can still estimate task finish time and avoid some unnecessary speculative tasks. Since non-running tasks will be schedule first, time consumed by speculative tasks take less portion of total execution time. We analyze this effect by adjusting the ratio of total task number over the number of TaskTrackers. If the number of tasks becomes larger, the ratio becomes bigger and there are more waves of tasks. If the number of tasks becomes smaller, the ratio becomes smaller and there are fewer waves of tasks. We use the same scenario as 4.1, and we change the mapper number to observe the effect on LA. Experimental results are shown in Figure 4.4. Though we did successfully avoid some speculative tasks, we do not have that much of improvement when ratio of total task number over the number of TaskTrackers becomes larger. Reasons are explained above.

Chapter 5

Conclusion and Future Work

MapReduce programming framework provides a new way for programmers dealing with large datasets. Programmers who use MapReduce framework need not to worry about the detail of task distribution, scalability and fault tolerance. MapReduce framework handles all those details in order to make programmers focus on their high-level algorithm. Current MapReduce framework have not yet forms a perfect scheduler. New features and flaws are presented by researchers and developers. We have discovered some flaws about how MapReduce framework schedules tasks. We propose a scheduling scheme for MapReduce to improve the utilization by avoiding unnecessary speculative tasks. Our scheduler can be easily merged into MapReduce framework and we implement it on Hadoop MapReduce. By scheduling on jobs demand and avoiding unnecessary speculative tasks, we can improve the utilization up to 25%.

Good estimation of a computing task is an important factor to our scheduler. Estimation of different attributes can be discussed and studied in the future. Hadoop MapReduce uses heartbeats as the task assigning/reporting mechanism. Short jobs that take only few minutes can lose large fraction of time waiting for heartbeats. Dynamic heartbeat interval adjustment can also be another worthy topic to study, further improving the performance of MapReduce.

Bibliography

- [1] J. Dean, S. Ghemawat, and G. Inc, “Mapreduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, 2008.
- [2] A. Thusoo and N. Jain, “Facebook’s petabyte scale data warehouse using hive and hadoop.” [Online]. Available: <http://www.infoq.com/presentations/Facebook-Hive-Hadoop>
- [3] R. Varela, “Slide of introduction to data processing using hadoop and pig.” [Online]. Available: <http://www.slideshare.net/phobeo/introduction-to-data-processing-using-hadoop-and-pig>
- [4] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
- [5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.
- [6] J. Dean, S. Ghemawat, and G. Inc, “Mapreduce: simplified data processing on large clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, 2004.
- [7] C. tao Chu, S. K. Kim, Y. an Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, “Mapreduce for machine learning on multicore,” in *Proceedings of the 20th Annual Conference on Neural Information Processing Systems*, 2006.

- [8] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *Proceedings of the 8th IEEE International Conference on Data Mining*, 2009.
- [9] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," in *Proceedings of the 35th International Conference on Very Large Data Bases*, 2009.
- [10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murth, "Hive - a warehousing solution over a map-reduce framework," 2009.
- [11] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.
- [12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.
- [13] "Apache hadoop project." [Online]. Available: <http://hadoop.apache.org>
- [14] "Hadoop fairscheduler." [Online]. Available: http://hadoop.apache.org/common/docs/current/fair_scheduler.html
- [15] "Hadoop capacityscheduler." [Online]. Available: http://hadoop.apache.org/common/docs/current/capacity_scheduler.html
- [16] M. Zaharia, A. Konwinski, A. D. Joseph, Y. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.
- [17] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *ACM Queue*, vol. 6, no. 2, 2008.
- [18] Z. Dadan, W. Xieqin, and J. Ning kang, "Distributed scheduling extension on hadoop," in *Proceedings of the 1st IEEE International Conference on Cloud Computing*, 2009.

- [19] Y. H. C. Tian, H. Zhou and L. Zha, "A dynamic mapreduce scheduler for heterogeneous workloads," in *Proceedings of the 8th International Conference on Grid and Cooperative Computing*, 2009.
- [20] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)," RFC 1157, Internet Engineering Task Force, May 1990. [Online]. Available: <http://www.ietf.org/rfc/rfc1157.txt>
- [21] K. McCloghrie and M. Rose, "Management Information Base for network management of TCP/IP-based internets," RFC 1156, Internet Engineering Task Force, May 1990. [Online]. Available: <http://www.ietf.org/rfc/rfc1156.txt>
- [22] "Net-SNMP Project." [Online]. Available: <http://net-snmp.sourceforge.net/>
- [23] "Hadoop project issue number 5784." [Online]. Available: <https://issues.apache.org/jira/browse/HADOOP-5784>
- [24] "Improve performance on small hadoop clusters." [Online]. Available: <http://pero.blogs.aprilmayjune.org/2009/11/30/improve-performance-on-sm%all-hadoop-clusters/>
- [25] "Hadoop map/reduce issue number 961." [Online]. Available: <https://issues.apache.org/jira/browse/MAPREDUCE-961>
- [26] "Hadoop map/reduce issue number 220." [Online]. Available: <https://issues.apache.org/jira/browse/MAPREDUCE-220>