UNIVERSITY OF AGDER

Time estimation for large scale of data processing in Hadoop MapReduce scenario

Li Jian

Supervisors
Anders Aasgaard , Jan Pettersen Nytun

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education.

University of Agder, 2011 May
Faculty of Engineering and Science
Department of Information and Communication Technology

Abstract

The appearance of MapReduce technology gave rise to a strong blast in IT industry. Large companies such as Google, Yahoo and Facebook are using this technology to facilitate their data processing[12]. As a representative technology aimed at processing large dataset in parallel, MapReduce received great focus from various organizations. Handling large problems, using a large amount of resources is inevitable. Therefore, how to organize them effectively becomes an important problem. It is a common strategy to obtain some learning experience before deploying large scale of experiments. Following this idea, this mater thesis aims at providing some learning models towards MapReduce. These models help us to accumulate learning experience from small scale of experiments and finally lead us to estimate execution time of large scales of experiment.

Preface

This report is the master thesis report in spring semester 2011. The topic derived from Devoteam, a well-known IT consulting company in south Norway. The main goal of this thesis is to discover how we can use cloud computing effectively. We choose the representative technology MapReduce as our research target.

Both cloud computing and MapReduce are complicated concepts. Through one semester's study and experiment, we came up with some learning models to make MapReduce serve us in a better way. MapReduce is truly a useful and practical technology to process large scale of data. It can solve some complicated enterprise level problems in an elegant way.

Last but not the least,I would like to thank my supervisors Anders Aasgaard and Jan P. Nytun for their careful and patient supervision. Due to their efforts, I got a lot of constructive ideas in doing experiments and academic report writing. Big thanks to them.

Grimstad Norway
January 2011
Li Jian

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The advent of cloud computing has triggered a tremendous trend on its research and products development. Basically, it is a network-based computing [5]. It combines technologies such as virtualization, Distributed File System, Distributed computing and so on to make it a very powerful resource pool. As a representative technology in cloud computing era, MapReduce also attracted great attention from both industry and academic institutions. Simply speaking, MapReduce is a programming model targeted at processing large scale of data in a parallel manner. "Large scale" refers to a dataset that may exceed terabyte or even petabyte. Facing with such a huge dataset, how to estimate its execution time under certain amount of hardware and how to customize just-enough hardware under certain bottleneck is quite important . As data size and computing nodes grow, how is the execution time curve like and how we tune our program to make it predictable are also important questions.

1.1 Problem definition

Based on the knowledge above, we describe our research problem to be estimating the execution time for large scale of dataset processing through small scale of experiment. Namely, given a huge dataset and sufficient machines, we aim to estimate its execution time by sampling a part of data and using a part of available machines.

Basically, our work is a practice of learning theory. We learn from completed experiments and apply this learning experience to new experiments. Our main problem generates the following subproblems:

- How does MapReduce parallelize a problem?
- What effect comes into being once we increase total data size and computing nodes? And how does MapReduce handle these effect?
- What approximation and assumptions do we need to achieve our estimation?

In parallel computing, communication and data synchronization are complicated problems, so it is the same with scheduling and coordination. Unlike traditional parallel programming model MPI(Message Passing Interface) [21], MapReduce shifts its focus from parallelizing a problem via point-to-point communication to designing basic task unit. How MapReduce simplifies parallel programming will be illustrated in Chapter 3. Further more, what factors may bring about bottleneck for parallelism will also be revealed.

1.2 Research goals and facilities

Our main goal is to propose convenient models to learn Hadoop MapReduce effectively. These models guide us on what to learn and how to apply learning experience on new experiments.

We use Amazon's cloud service Elastic Computing Cloud(EC2)[7] to accomplish our experiments. EC2 is an infrastructure service which provides virtual machines for customers. All virtual machines we use are 1.7 GB of memory, 1 EC2 Compute Unit,which is equivalent to 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor [15]. Except virtual machine's performance, Amazon also has guarantee on the network connection and point-to-point bandwidth [3].

We use on-demand virtual servers from Amazon. This service allows us to terminate virtual servers immediately after we finish our computation tasks. Standing at customer's view, it saves cost. But one problem is each time we start a new cluster environment, the network topology can not stay the same with the previous network. As servers are virtualized, network is also virtualized, therefore changes to virtual servers introduces change to network. Because of various concerns, cloud providers normally prohibit customers probing network topology. But their ensurance on point-to-point bandwidth provides a good reason to divert customers' attention away from this aspect.

1.3 Thesis outline

Up till here, we have described research problem and experiment environment in detail. Chapter 2 will mainly introduce necessary theories that will help us solve our research problem. Chapter 3 gives a detailed introduction about our research target Hadoop MapReduce. It's mechanism will be elaborated. In Chapter 4, we deepen our understanding of Hadoop MapReduce by proving a key proposition. Then we propose two models to solve our research problem. Chapter 5 is Discussion chapter, which gives a detailed analysis of our experiment results. Chapter 6 is Conclusion chapter, which can be seen as a summary of our research work.

1.4 Constraints and limitations

MapReduce is a network based programming model. The more computing nodes, the more complex the network is. In order to simplify our problem, we limit computing nodes to be less than 1024. The reasons that we choose 1024 as our boundary include:(1) According to IEEE 802.3i standard(10BASE-T)[14] , a star network can accommodate 1024 nodes. (2) Star network is simple to achieve and a network of 1024 nodes can do a great many things for an enterprise. Further more, Yahoo has experimented Terabyte sort program on more than 1406 nodes[24]. As of 15 May 2010, Yahoo has used 3452 nodes to finish 100TB's sort within 173 minutes[10]. Because most of our research is based on star network, we suppose limiting the number of nodes to be 1024 is necessary and reasonable.

This thesis is not working on actual enterprise problems. Therefore, the experiment data is generated by some tools. Because of budget limitation, we are not able to work on a real "huge" dataset and a very big number of machines. We just use our current experiment and theoretical analysis to prove our models and other conclusions are reasonable.

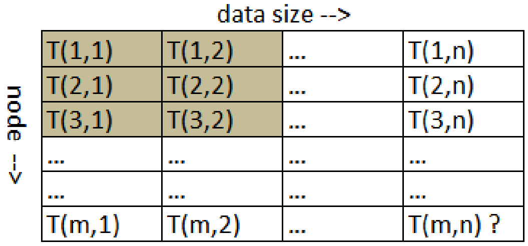
Since we use virtual servers from Amazon, the network is maintained by Amazon. Due to the limitation set by Amazon ,we couldn't configure proper experiment environment for some special situations. In other words, our experiments can only support a part of analysis and conclusions. But we will use proper assumptions, math and other knowledge to support other analysis and conclusions.

Chapter 2

Prior research

2.1 Methodology

Before stepping into our problem, we present a further problem abstraction. This abstraction is shown in the following Figure 2.1. Right now, our problem becomes experimenting the dark area in Figure 2.1 and then estimating $T(m, n)$, where m represents the number of computing nodes and n denotes data size, eg. we can define $n = 1$ as 1GB's data.



data size -->

node -->

T(1,1)	T(1,2)	...	T(1,n)
T(2,1)	T(2,2)	...	T(2,n)
T(3,1)	T(3,2)	...	T(3,n)
...	...	...	...
...	...	...	...
T(m,1)	T(m,2)	...	T(m,n) ?

Figure 2.1: Problem abstraction.

Figure 2.1 gives us hint that execution time T can be expressed as a function $T(nodes,$ $size)$, where variable $nodes$ stands for the number of parallel nodes and $size$ stands for the size of input data . If $T(size, nodes)$ can be generated through small scale of experiment, and most importantly the trend of T is regular, then we can use it to estimate execution time given a certain $size$ and $nodes$. Our strategy is to fix $nodes$ and then compute $T(size)$. Similarly, we attempt to get $T(nodes)$ by fixing $size$. Finally, we combine these two functions together to form $T(size, nodes)$.

In fact, $T(size)$ here is another format of time complexity. $T(nodes)$ measures when $nodes$ changes how execution time T changes. This change is often described by *speedup*.

We assume readers to this thesis already have some knowledge about time complexity. Therefore, we only introduce speedup theory in the following section.

2.2 Parallel computing

Parallel computing aims at shortening the execution time of a problem by distributing computation tasks to multiple computing nodes. Parallel algorithms must be designed to make it work. The designers must consider communication, load balancing, scheduling and data synchronization across nodes. Comparing with sequential execution, these complicated problems often cause headache for designers. How Hadoop MapReduce solves these problems will be covered in Chapter 3.

The performance of a parallel system is measured by speedup. Mature theories such as Amdahl's Law[1] and Gustafson's Law[11] provide useful guidance for our research. Similar to any parallel architectures, data synchronization is also a big problem for Hadoop MapReduce. The next few sections will cover speedup,data synchronization and other important issues in detail.

2.2.1 Common network topologies

Parallel computing can be classified into many categories. A distributed cluster environment is a way of achieving parallel computing, where each node has independent processor and memory and they communicate through network. Data exchange is a very frequent activity over the network. Therefore, network topology plays a very important role for cluster-based parallel computing.

In order to facilitate the following description, we emphasize an important concept: *point-to-point bandwidth*. It describes the transmission speed between one computer and another computer.

Topologies such as star network, tree network and mesh network can be used for parallel computing. The simplest star network showed in Figure 2.2 is that each node is directly connected to a central device such as a switch or router but no pair of nodes has direct connection. In such a network, data communication between nodes is handled by a central device. Its performance decides the performance of the network.

In fact, data centers often organize a group of servers on a rack and one rack normally

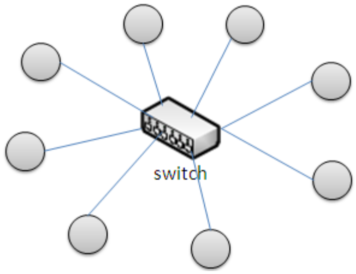


Figure 2.2: A star network:each circle represents a node.

contains 30-40 servers. One rack has a centralized switch in connection with those servers. This topology on a rack is a star topology. Different racks are connected via anther level of switch. Thus several star networks compose a tree network, which is shown in Figure 2.3.

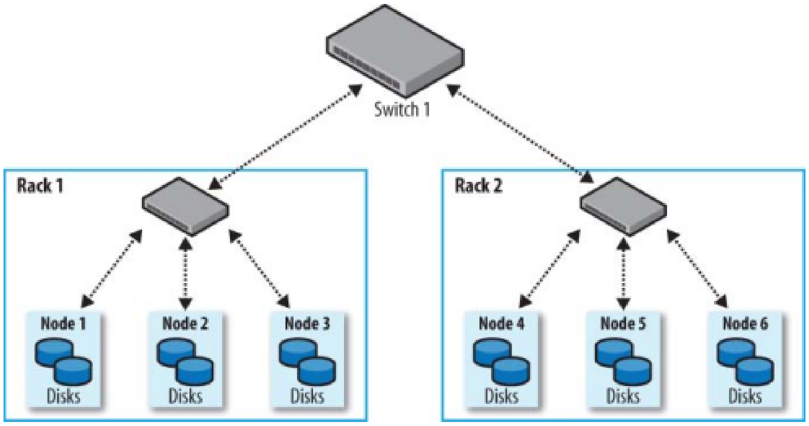


Figure 2.3: Typical two-level tree network for a Hadoop Cluster [30]

Considering this two-level tree network, the point-to-point bandwidth available for each of the following scenarios becomes progressively less[30]: (1) Different nodes on the same rack; (2) Nodes on different racks in the same data center.

For an infrastructure owner, he can build a star, mesh or tree network according to his will. But in a public cloud[1], different vendors may adopt different topologies. Due to various concerns such as security, commercial secrets and so on, they often don't let customers know what exact physical topologies they use. For example, Amazon Web Service allows users to build a virtual cluster, but it is banned to trace its network topology. Amazon just has a guarantee to customers that it ensures a certain point-to-point bandwidth. How the bandwidth is allocated and its relationship to network topology are unknown to customers.

[1] Public cloud is a form of cloud computing that vendors provide computing service to customers but hide technology details.

2.2.2 Granularity

Granularity means the size of a basic operation unit that we take into consideration. Coarse-grained systems consist of fewer, larger units than fine-grained systems [9]. For a parallel algorithm, a basic operation such as addition, subtraction and multiplication can be seen as a basic computing unit. For data parallelism, the size of evenly partitioned data unit can serve as a granularity unit. The larger a data split is, the coarse-grained it is.

2.2.3 Speedup

Speedup is an important measurement criteria to the performance of a parallel system. It measures how faster a parallel system performs than a sequential execution scheme[8]. It is widely applied to describe the scalability of a parallel system [17, 29]. Amdahl's Law[2] provides a method to compute speedup by workload, but it can also be computed through execution time. This section gives a detailed introduction of speedup and its relationship with our research.

2.2.3.1 Compute speedup by execution time

Speedup is a concept that only applies to parallel system. Its original definition is the ratio of sequential execution time to parallel execution time[8] which is defined by the following formula:

$$S(n) = \frac{T(1)}{T(n)} \tag{2.1}$$

where:

- n is the number of processors. We assume one node has one processor, thus the number of computing nodes equals processors.
- $T(1)$ is sequential execution time, namely using one node. T(1) is the reference standard or baseline to compute speedup.
- $T(n)$ is the parallel execution time when running n nodes.

Theoretically, $T(1)$ is a necessary component to compute speedup, however, as far as enormous computation work is concerned, we normally don't do such a heavy experiment on a single machine. There are two strategies to handle this: (1) use a bundle of machines and take the bundle as baseline; (2) estimate $T(1)$ by small scale of experiment on one node.

Considering several consecutive phases and each phase is parallelizable, we should compute speedup for each phase separately and then merge them to be the final speedup. Denote T_{pi} as execution time for the ith phase, we can use the following equation to compute speedup:

$$\begin{aligned} \text{if} \quad & T(1) = T_{p1} + T_{p2} + ... + T_{pk} \\ \text{then} \quad & T(n) = \frac{T_{p1}}{S_1(n)} + \frac{T_{p2}}{S_2(n)} + ... + \frac{T_{pk}}{S_k(n)} \\ & S(n) = \frac{T(1)}{T(n)} = \frac{T_{p1} + T_{p2} + ... + T_{pk}}{\frac{T_{p1}}{S_1(n)} + \frac{T_{p2}}{S_2(n)} + ... + \frac{T_{pk}}{S_k(n)}} \end{aligned} \tag{2.2}$$

Equation 2.2 tells us the final speedup is computed through subcomponents $(S_1(n), S_2(n), .., S_k(n))$. We can view the vector $(T_{p1}, T_{p2}, .., T_{pk})$ as the weight of vector $(S_1(n), S_2(n), .., S_k(n))$.

In order to enhance our readers' understanding, we provide an example in Figure 2.4[1]. Assume part A and B must be executed sequentially, part A takes 80% of the whole time and part B only occupies 20%. If we accelerate part A 4 times and part B 2 times ,the final speedup is $(0.8 + 0.2)/(\frac{0.8}{5} + \frac{0.2}{2}) = 3.8$. From this example we see that, the larger the weight of a phase is, the more influence it has on the final speedup.

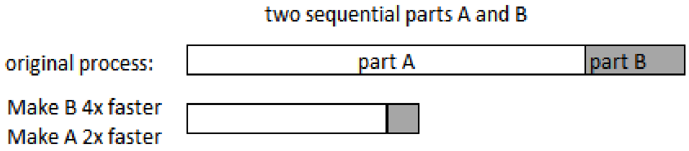


Figure 2.4: Speedup computation example[1]

2.2.3.2 Compute speedup by Amdahl's Law

Functioning the same as the formula 2.1, Amdahl's Law defines speedup from another angle. This law describes speedup S as follows:

$$S = \frac{W_s + W_p}{W_s + W_p/n} \tag{2.3}$$

where:

- n is the number of computing nodes;
- W_s represents the non-parallelizable work of the total work.
- W_p refers to the parallelizable work of the whole work;

The key to understand Amdahl's Law is to make sense of W_s and W_p. Figure 2.5 shows their difference.

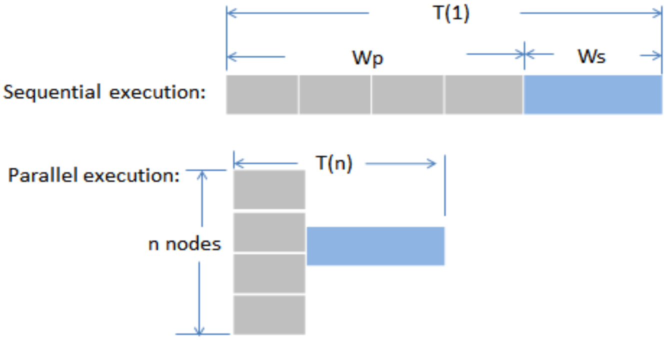


Figure 2.5: Parallelizable and non-Parallelizable work

Amdahl's Law has a different form with Formula 2.1, but in fact they are consistent. Assume each processor has the same processing speed V, then the $T(1) = \frac{W_p+W_s}{V}$ and $T(n) = \frac{W_p/n}{V} + \frac{W_s}{V}$. Replacing $T(1)$ and $T(n)$ in Formula 2.1 naturally derives Amdahl's Law.

The importance of Amdahl's Law is that it classifies work to be parallelizable work and non-parallelizable work. This strategy further reveals the upper bound of a parallel system. As n approaches infinity, S approaches $\frac{W_s+W_p}{W_s}$, which is the maximum speedup of a system[2]. On the other hand, if we know the ratio of parallelizable work to the whole work, the maximum speedup can be calculated. For instance, if parallelizable work occupies 95% of the whole work, then maximum speedup is 20. Figure 2.6 shows Amdahl's Law lays limitation on speedup.

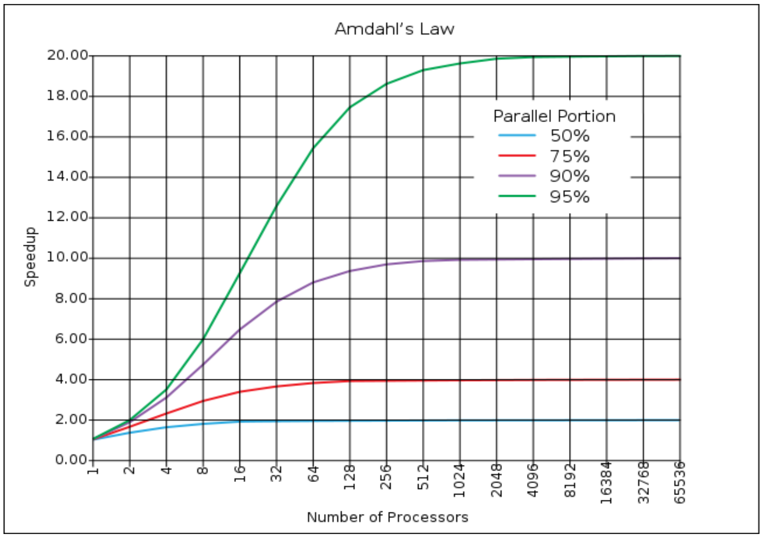


Figure 2.6: Amdahl's Law limits speedup [1]

2.2.3.3 Scaling behavior

Generally speaking, there are three types of scaling behaviors. They are super-linear scaling, linear scaling and sub-linear scaling. These three scaling behaviors are described in Figure 2.7. They actually described three speedup patterns. Speedup is calculated by he gradient of each line in Figure 2.7. Super-linear scaling means speedup is growing up, and sub-linear scaling means speedup is decreasing and linear scaling means speedup is kept the same.

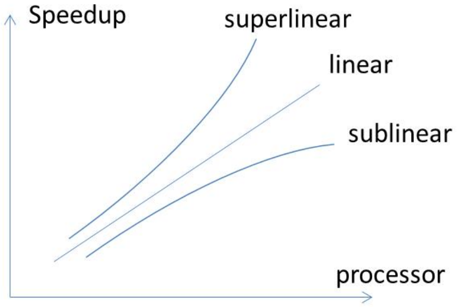


Figure 2.7: Scaling behavior

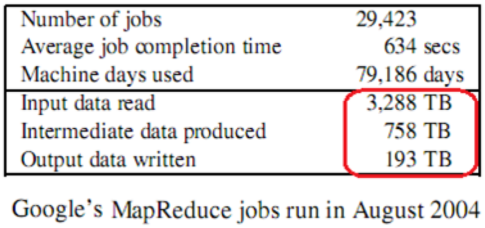
2.2.4 Data synchronization

Data synchronization aims at keeping consistency among different parts of data[28]. We use traditional WordCount algorithm to illustrate this. Assume there is a huge dataset, the size of which is much greater than the memory of our available computer. In a typical traditional WordCount algorithm, we make key/value pair $(word_i, 1)$ for $word_i$ that we encounter for the first time. Then we store it on hard disk. When we meet $word_i$ for the second time, we query the position of $word_i$ on hard disk and then load it to memory and plus one to its value. Thus its key/value pair becomes $(word_i, 2)$. Similarly, the kth time we meet $word_i$, its key/value pair becomes $(word_i, k)$. That is to say, pairs $(word_i, value1)$ and $(word_i, value2)$ are put together and merged to be $(word_i, value1 + value2)$. In this scenario, placing the pairs with the same key together is data synchronization. It helps us merge data. The problem with this traditional algorithm is there are too many queries. This data synchronization gives rise to a large amount of workload. In Hadoop MapReduce, how this heavy workload is relived is covered in Chapter 3 Section 3.1.2 Job execution workflow.

Chapter 3

Research target:Hadoop MapReduce

MapReduce is programming model aimed at large scale of dataset processing. It originated from functional language Lisp. Its basic idea is to connect a large number of servers to process dataset in a parallel manner. Google started using this technology from the year 2003. Thousands of commodity servers can be connected through MapReduce to run a computation task. Figure 3.3 shows Google's use of MapReduce in 2004.



Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB

Google's MapReduce jobs run in August 2004

Figure 3.1: Google use mapreduce

3.1 Mechanism

The execution workflow of MapReduce has been elaboratively described in Jeffrey Dean's paper [6]. Although it covered the core workflow, it didn't cover the architecture and deployment. Therefore, we choose a bottom-to-top strategy to show the whole view of Hadoop MapReduce. However, we should emphasize that Google MapReduce and Hadoop MapReduce are not the same in every aspect. As an open source framework inspired by Google MapReduce, Hadoop MapReduce implemented most of the characteristics described in Jeffrey Dean's paper [6], but it still has some features of its own.

3.1.1 Job hierarchy

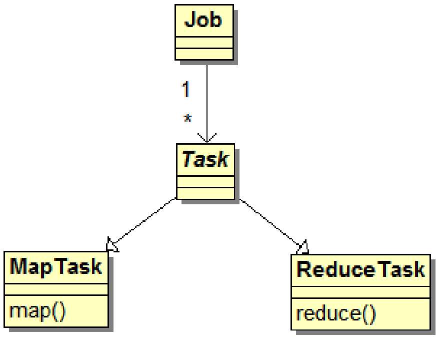


Figure 3.2: Job Hierarchy

map: $(k_x, v_x) \rightarrow$
$(k_{x1}, v_{x1}), (k_{x2}, v_{x2}), (k_{x3}, v_{x3})\ldots$

reduce: $(k_y, [v_{y1}, v_{y2}, v_{y3}]) \rightarrow (k_y, v_y)$

Figure 3.3: map and reduce function [6]

Figure 3.2 shows the sketch of a basic working unit *job* for MapReduce. A *job* is similar to the term *project* in C++ or Java development environment. As shown in Figure 3.2, a *job* contains multiple *tasks*, and each *task* should either be a *MapTask* or *ReduceTask*. The functions *map()* and *reduce()* are user-defined functions. They are the most important functions to achieve a user's goal.

Map function transforms a key/value pair into a list of key/value pairs; while reduce function aggregates all the pairs sharing the same key and processes their values. Figure 3.3 shows the function of map and reduce.

The combination of map and reduce simplifies some traditional algorithms. We use the program WordCount to illustrate this. By default, the input of a map function is (key,value) pairs, in which key is line content of a text file, and value is line number. The map function of WordCount splits the line content to be words list, each word is key and its value is 1. Then reduce function aggregates all the pairs which have the same key and adds their values together. The sum of these values sharing the same key is the total number of its correspondent word. The workflow of WordCount is shown in Figure 3.4.

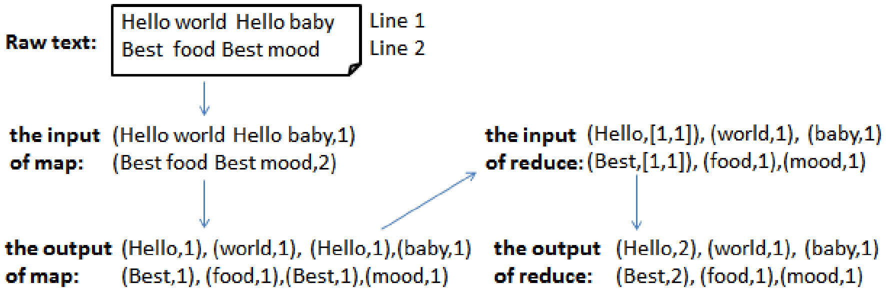

Raw text:
Hello world Hello baby
Best food Best mood
Line 1
Line 2
the input of map:
(Hello world Hello baby,1)
(Best food Best mood,2)
the output of map:
(Hello,1), (world,1), (Hello,1),(baby,1)
(Best,1), (food,1),(Best,1),(mood,1)
the input of reduce:
(Hello,[1,1]), (world,1), (baby,1)
(Best,[1,1]), (food,1),(mood,1)
the output of reduce:
(Hello,2), (world,1), (baby,1)
(Best,2), (food,1),(mood,1)

Figure 3.4: The workflow of WordCount

3.1.2 Job execution workflow

Based on the knowledge of map and reduce function, now let's look at their execution workflow in a cluster environment. Figure 3.5 shows Google's MapReduce cluster execution workflow. Hadoop MapReduce's workflow is the same with it.

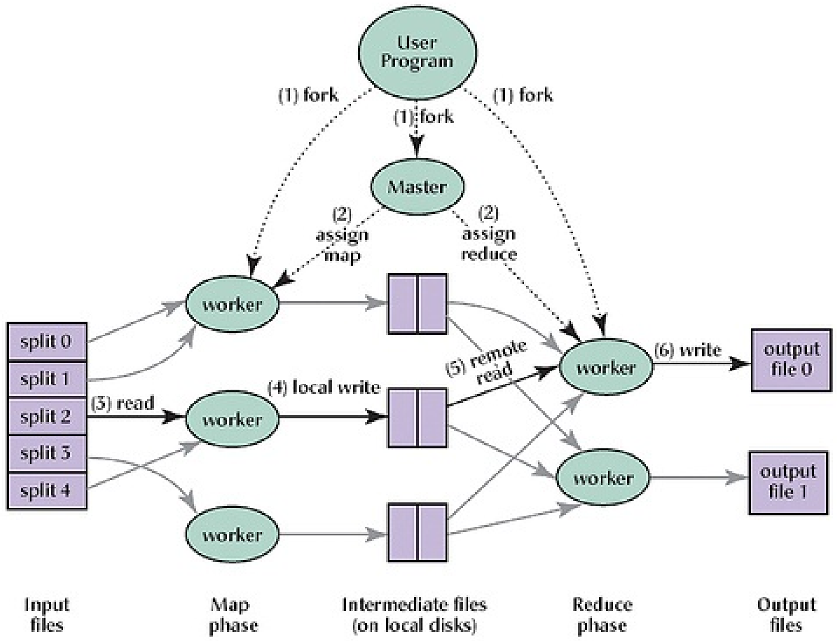


Figure 3.5: MapReduce cluster execution workflow [6]

Jeffrey Dean's paper [6] illustrated this workflow elaboratively. Here We just interpret it by decoupling this workflow in our own words. Nevertheless, our logic is similar to Jeffrey Dean. Since Hadoop MapReduce was inspired by Google MapReduce, here we just use the mechanism of Google MapReduce to illustrate Hadoop MapReduce.

Before we start out, it is necessary to explain the key words appeared in Figure 3.5:

master Master is a computer that handles the management work such as scheduling tasks and resource allocation for tasks. There is only one master node in a Hadoop cluster, and the other nodes serve as slave nodes. Sometimes, we use one node to be a backup node for master. The backup node doesn't participate any actual management job. It only backup the most important information for the running master node.

fork It is the process of distributing user program to all machines in the cluster.

split A big file is divided into many splits and each split is usually customized to be 64MB. Splits facilitates data parallelism and backup. They are distributed to slave nodes for processing. Each split can be replicated to other machines as backup.

worker The worker in the figure is a logical concept, not a physical machine. A worker has the same meaning with a task. A node can have many workers to execute basic map or reduce operations. A map worker is also named as a mapper or a map task and a reduce worker is named as a reducer or a reduce task.

Basically, a job contains two sequential phases:map and reduce phase. The map phase in Figure 3.5 is simple. It takes file splits as input and writes intermediate results to local disks. The reduce phase is a bit complicated. Mappers and reducers have many-to-many relationship. One reducer fetches data from all mappers; and one mapper partitions its output to all reducers. This action is performed by partition function. In Hadoop, partition function is allowed to be overwritten by users . Partition function plays the role of a connector between map and reduce phase. It decides which key goes to which specific reducer and achieves data synchronization. The partition function for WordCount program is $hash(key)\ mod\ R$,where R is the number of reducers. This partition function achieves data synchronization in an elegant way. It doesn't need any query to put the same word to the same reducer.

The result of partition function is a reducer gets a lot of pairs $(word_i, 1)$ and the same pair $(word_i, 1)$ may appear multiple times. For example, a reducer might get such pairs sequentially: $(word_i, 1), (word_j, 1), (word_i, 1), (word_k, 1)....$ In Hadoop MapReduce, a reducer doesn't merge pair $(word_i, 1)$ with its previous occurrence immediately after it fetches one more $(word_i, 1)$. The strategy is after a reducer obtains all its pairs and then it sorts all of them by key so that the pairs sharing the same key are naturally grouped together.

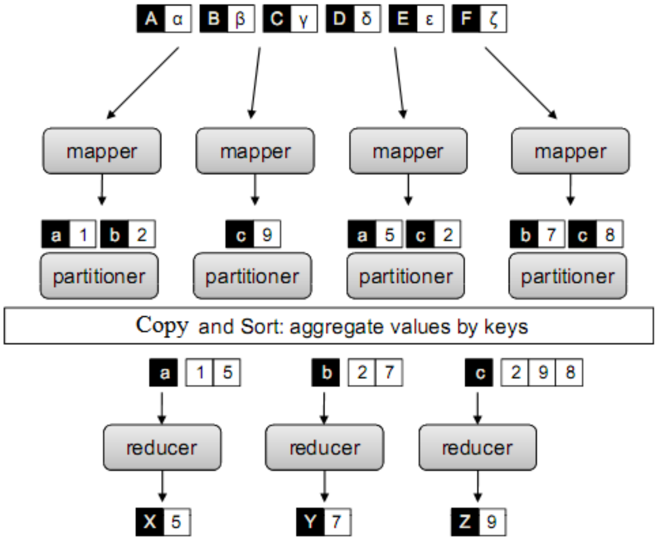


Figure 3.6: MapReduce partitioner[18]

The details of partition function and reduce phase is explained via Figure 3.6. This figure shows that partitioner is executed in between a mapper and reducer. It is executed in the same machine with its mapper. Partitioner is a bridge between a mapper and a reducer. In Figure 3.6, "copy[1]" means reducers fetch intermediate results from mappers though Http protocol. "Sort" achieves aggregation for reducers. In Hadoop MapReduce, each reduce task has three subphases:copy, sort and reduce. The subphase reduce fulfils user-defined function. In WordCount program, the reduce operation is to sum the values sharing the same key.

Notice that we use "reduce phase","reduce tasks(reducers)", "subphase reduce(or subreduce)" and "reduce operation(reduce function)" to distinguish their differences. Similarly, "map phase","map tasks(mappers)" and "map operation(map function)" are also used to differentiate them. The hierarchy and multiplicity relationship among them are important. Figure 3.7 shows the relationship among these reduce-related concepts.

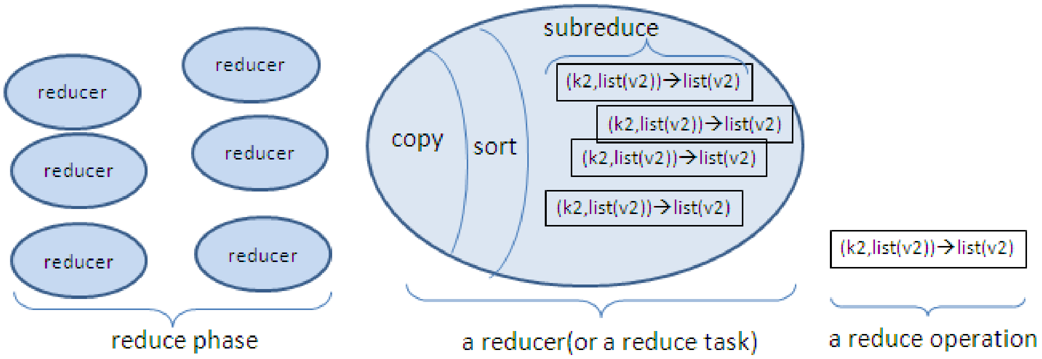


Figure 3.7: Disambiguate concepts in reduce phase

3.1.3 Hadoop cluster hierarchy

Figure 3.8 shows Hadoop cluster hierarchy. It shows us how a job and its tasks are arranged among physical machines. Master node is also named as job tracker, which manages all the nodes. It needs an IP address table containing all IP addresses of all nodes. The slave nodes are also called datanode or task tracker. They contribute their local hard disk to form a Hadoop Distributed File System(HDFS). They process data directly. From the perspective of a master node, all the slave nodes are dummy nodes. Except configuring slave nodes to be able to run Hadoop MapReduce program, we just use master node to control job execution and manage slaves.

[1] The operation is actually moving data, rather than copying data. In order to be in line with the description of Hadoop's official documents, we still use the term copy.

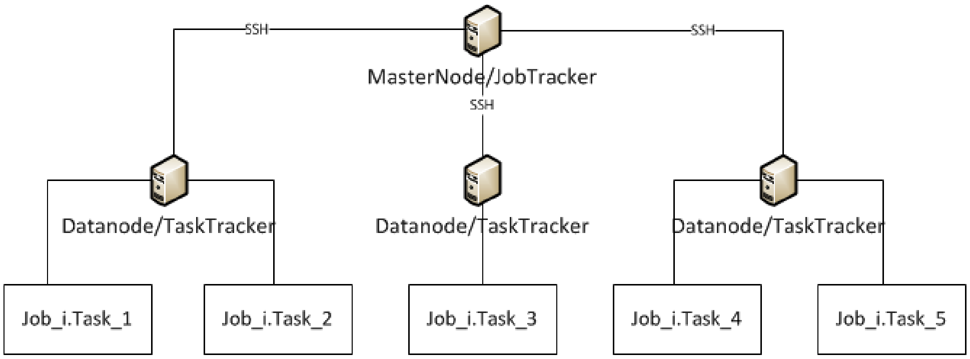
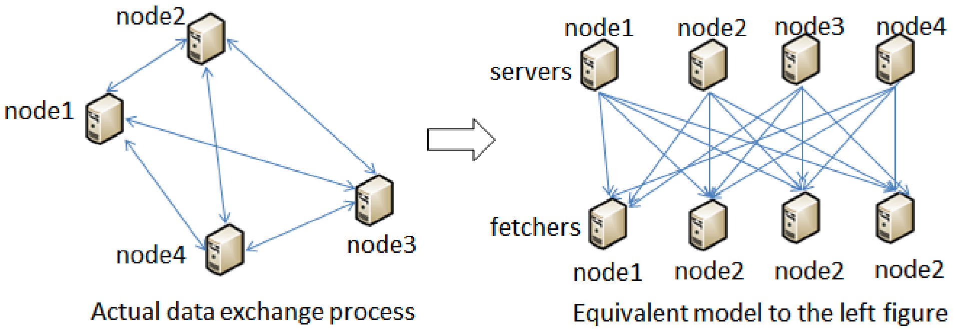


Figure 3.8: Hadoop cluster hierarchy

3.1.4 Traffic pattern

In parallel computing, data exchange is inevitable, therefore making sense of traffic pattern is of great importance. Large scale of data exchange over the cluster often reveals some bottlenecks. In Hadoop MapReduce, large scale of data exchange happens after the completion of map phase. This data exchange process refers to subphase “copy” in Figure 3.6. From the perspective of a reducer, it is executed on an individual node but its input data is constructed from multiple mappers. From the perspective of a mapper, it partitions data to multiple reducers and it is likely to produce a part of input data dedicated to any reducer. Thus, from the perspective of tasks, data transmission between mappers and reducers is a many-to-many relationship. Correspondingly, from the perspective of computing nodes, it is a multipoint-to-multipoint transmission relationship, because every node contains both mappers and reducers. Thus, if all nodes are executing copy operation simultaneously, every node is transmitting data bidirectionally. Under this situation, the reducer of a node is downloading data from all nodes including itself, and the mappers of that node are uploading their results to all nodes. Therefore, referring to Figure 3.5 we see that one mapper spills data to multiple reducers and one reducer fetches data from multiple mappers. This bidirectional multipoint-to-multipoint data exchange process is described in Figure 3.9.



Actual data exchange process

Equivalent model to the left figure

Figure 3.9: Traffic pattern:servers and fetchers are the same set of nodes.

After map phase is finished, data exchange is not fulfilled once and for all. Every reducer always follows such an execution seqence:copy and then sort and finally reduce. In other words, copy is bound with sort and reduce operation so that copy operation is executed intermittently. Between one reducer's copy operation and its subsequent reducer's copy operation, there is a time interval for sort and reduce operation. Figure 3.10 provides a graphical description of this intermittent copy. We emphasize this intermittent copy because in previous Hadoop version pre-0.18.0[23], it is once-and-for-all copy. Without explicit explanation, it is very confusing.

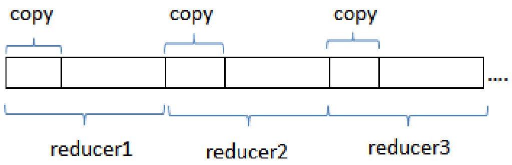


Figure 3.10: Intermittent copy

Even though we mentioned it's a multipoint-to-multipoint data exchange process, the data fetching pattern is not very complicated. We view this data fetching activity from two perspectives: an individual reducer and a set of parallel reducers.

Hadoop has a property *mapred.reduce.parallel.copies*, which is set to be 20 by default. It means that a reducer can fetch data from 20 mappers simultaneously. This value is fixed during the execution of a MapReduce job. Therefore, if the total number of mappers is larger than 20, from the perspective of an individual reducer, its data fetching can be seen as a sequential process. Thus, once the input data size of a reducer is fixed, no matter the input data is constructed from n nodes or $2n$ nodes, the number of mappers to that reducer is almost the same and the data volume from every mapper of them also doesn't change, thus the fetching time is the same. For simplicity reason we ignore the time delay due to Http connection setup and teardown. It is obvious that fetching data from $2n$ nodes needs more Http connection setup and teardown than n nodes. From the perspective of a set of parallel reducers, data fetching is parallelized. One reducer's data fetching activity happens simultaneously with other reducers' data fetching.

From this introduction we see that Hadoop lays more focus on logical entities such as mappers and reducers than physical entities computing nodes. According to common knowledge, the total input data is more associated to the number of mappers than the number of nodes, thus removing the concern about the number of nodes simplifies our problem. Facing with the same problem, the number of mappers will not change even if the number of nodes increases. That is to say, a MapReduce program is adaptive to different number of nodes without any necessity to change its code.

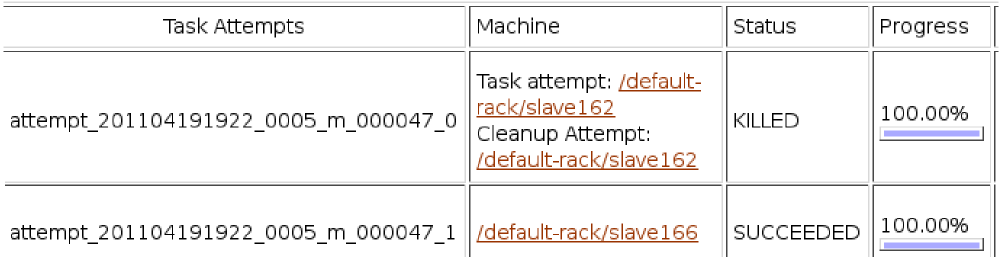
3.2 Supportive platform:Hadoop Distributed File System

MapReduce was originally designed to solve large dataset problems. The “large” dataset may exceed several terabytes. Storage is a challenge for such big datasets. Google designed Google File System(GFS) to solve this problem and accordingly Hadoop has Hadoop Distributed File System(HDFS). The details of GFS is unknown to public, here we just introduce HDFS. As Figure 3.8 shows, a lot of slaves are under the control of a master machine. The master uses Secure Shell(SSH) to start HDFS daemon on every slave. HDFS calculates the storage space on every slave and combine them to be a big storage system. Unlike traditional file system, HDFS doesn’t have partitions. The master node and any slave node can upload files to HDFS and download them. Similar to GFS, users can define the size of split, typically 64MB. This value 64MB means that an uploaded file is partitioned by the basic unit 64MB. Therefore, a big file usually have multiple splits.

In order to facilitate data fetching, HDFS distributes splits to all slaves. Local splits have higher processing priority than remote splits, so that in some cases, slaves do not have to fetch remote data to accomplish distributed computing. This mechanism reduces network burden for distributed computing. Further more, considering the failure of machines, HDFS allows users to define a replica value. Its default value is 3, which means every split has three copies(including itself), and they are also distributed over the network. Hence, the failure of a single machine does not cause data loss and computation failure for a computation task. In a word, HDFS is a scalable and fault-tolerant distributed file system.

3.3 Key feature:speculative execution

Speculative execution[27] is a feature of Hadoop MapReduce. It computes the execution progress of tasks and speculates whether they are slow. Slow tasks are re-executed by another node. This mechanism avoids slow tasks from lasting too long, and reduces the gap among tasks’ execution time. In other words, it boosts the balance of the execution time of tasks. An example of speculative execution is shown in Figure 3.11.



Task Attempts	Machine	Status	Progress
attempt_201104191922_0005_m_000047_0	Task attempt: /default-rack/slave162 Cleanup Attempt: /default-rack/slave162	KILLED	100.00%
attempt_201104191922_0005_m_000047_1	/default-rack/slave166	SUCCEEDED	100.00%

Figure 3.11: Speculative execution

3.4 Hadoop MapReduce program structure

Up till now MapReduce has many implementations. Its libraries have been written in C++, C#, Erlang, Java[19] and so on. Hadoop MapReduce is written in Java, hence we only provide the program structure for Hadoop MapReduce. Figure 3.12 shows its program structure.

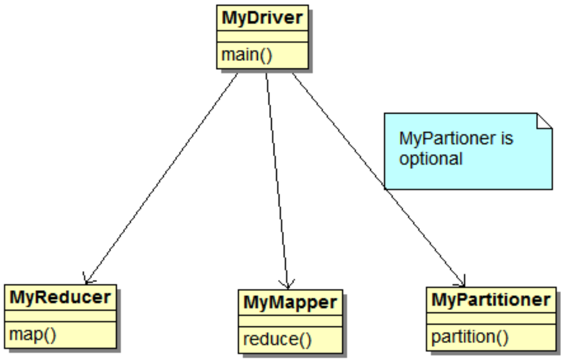


Figure 3.12: MapReduce program structure

Basically, Figure 3.12 is consistent with job hierarchy showed in Figure 3.2. The only difference is this figure is at the standpoint of a programmer. The entrance class is *MyDriver* class, in which a programmer defines which mapper/reducer class he uses. The whole workflow such as the number of total chained jobs and when to submit them is also defined in this class. Map and Reduce class rewrite the inherited method $map()$ and $reduce()$ to define how to process a task unit. Communication, coordination and task scheduling are embedded into Hadoop's library file. Thus, a programmer is exempt from this tough and heavy work. A mapper class is instantiated when it gets the command of processing one data split. In text processing programs, it's map method is invoked when it reads in a line. Normally, a mapper instance does not communicate with another map instance. Their status information is reported to master node and handled by embedded library. Namely, a programmer doesn't have to define which node processes which map tasks and he also doesn't have to define any communication rule for any two map tasks. These details about map task are the same as reduce tasks.

A sample WordCount program written in pseudo code is shown as follows:

Program 1 WordCount pseudo code

```
class MyDriver:
    main():
        Define a job
        set input/output format for that job
        Associate a mapper class to that job
        Associate a reducer class to that job
        Set the number of reducers
        Run this job

class MyMapper:
    map(String key,String value):
     //key:document line number
     //value:line content
        for each word w in value:
            store key/value pair (w,1);

class MyReducer:
    reduce(String word,Iterator values):
     //key:a word
     //values: a list of counts
        count=0;
        for each v in values:
            count=count+1;
        store key/value pair (word,count)
```

We need to emphasize that it is required to set the number of reducers, but not required to set the number of mappers. The number of mappers equals the number of splits. Up till now, the current Hadoop MapReduce is not smart enough to do everything for users. Users have to set the number of reducers according to their own requirement.

3.5 MapReduce applications

MapReduce is extensively used by Google, Yahoo,Facebook and so on. The application area mainly covers: distributed search, log batch processing ,sorting[6] and so on. We briefly describe some examples as follows:

Distributed search[6] The map operation returns a line when it matches a user-defined pattern. Since there is no use of reduce function, we don't have to set a reduce class and write its function.

Distributed sort A typical example is Terasort [23], where results are sorted by keys. The map phase does nothing but distributes data to reducers. Assume user sets N reducers. User's Driver program selects $N-1$ sampled keys from map and sort them. According to these $N-1$ sorted keys, a user-defined partition function partitions $(key, value)$ pairs to the intervals of $N-1$ sorted keys. This measure guarantees that the keys distributed to ith reducers are always within the range of sorted $(i-1)th$ key and ith key. The scenario is shown in Figure 3.13.

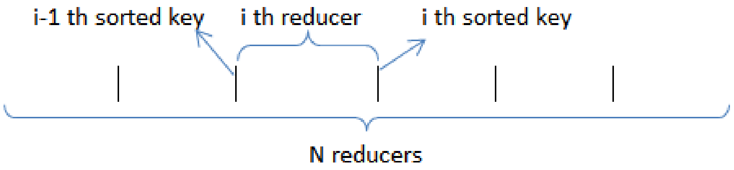


Figure 3.13: Distributed sort scenario: the vertical bar stands for a key.

File format batch conversion It's reported that New York times converted 4TB's scanned picture format files into PDF format [22]. This application only uses map phase to parallelize batch processing work.

3.6 Experiment plan

Previous research [25, 31, 18, 16, 12, 13] have customized several experiments on Hadoop MapReduce. These experiments include WordCount, matrix multiplication, Bayesian Classification, Sorting, Page rank ,K-means clustering and so on. Among all the experiments, WordCount and Sorting are widely adopted benchmarks [13]. We choose these two experiments to fulfil our needs. The reasons are that they are easy to parallelize by MapReduce and they are the basic ingredients for many other experiments.

Chapter 4

Proposed models and verification

According to the execution workflow of Hadoop MapReduce , we abstracted a more understandable model to describe it. This model is shown in 4.1. In our context, the term model refers to a simplified description of a problem, which leads to a solution directly. However, Figure 4.1 is an ideal situation, which must be based on certain assumptions and conditions. Therefore, it is important to clarify those important assumptions and conditions first. After that, we propose two models to estimate the execution time of a MapReduce job.

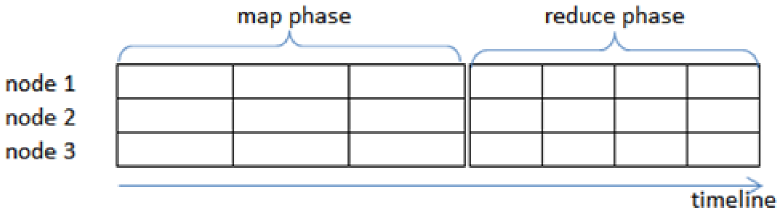


Figure 4.1: Simplified MapReduce workflow: a grid represents a task

4.1 Assumptions

Establishing a model is not an easy thing, questions like why a model is reasonable, what conditions it must satisfy and does it really match the real situation are very important. In order to remove those suspicion, we made some necessary assumptions and conditions in this section. The most important assumptions and conditions are listed as follows:

Homogeneous computing environment and stability Computers have the same performance. They are running the same type of operating system. The network is stable so that the bandwidth will not shrink during our program running time.

No severe workload imbalance We assume that each task handles almost the same workload. That is to say, the computation density is evenly distributed for data splits.

Granularity unit:a task A job may contain tens of thousands of map and reduce tasks, taking one map/reduce task as granularity unit can simplify our analyzing work [6]. In map phase, granularity unit is a map task and in reduce phase it is a reduce task.

Ignore management cost It is the master machine that mainly handles scheduling and other coordination work. We normally use a very powerful machine to be master machine, and its management work is normally far less than slaves' workload, therefore we assume master's management doesn't bring about explicit time delay for the whole job.

One node has one processor Current computers normally have multiple processors, we limit that one computer node has only one processor. Thus, the number of nodes is the same as the number of processors. This limitation just simplifies our explanation.

No node failure Hadoop MapReduce can handle node failures by diverting workload to other nodes. In our case, we don't consider node failure. We just use experiments that have no node failures for analysis.

Condition of approximate equality In the process of experiment verification, we are suppose to check whether value A and B are equal. For instance, A could be a measured value and B could be an estimated value. We use the following expression to judge whether they are equal:

$$|A - B| \leq \frac{1}{10} min\{A, B\} \tag{4.1}$$

4.2 Key condition:execution time follows regular trend

Before we propose our models, one critical proposition must be proved: the execution time of tasks obeys regular trend under certain conditions even if the number of nodes and the input data size change. We focus on the influence on two aspects: (1) the behaviors of an individual node, (2) the behaviors of a set of parallel nodes.

Individual behavior mainly refers to the time consumption of the same type of tasks within an individual node. In terms of group behavior, we mainly focus on the following two aspects:

- When the number of nodes increases, is there any change to the parallel behavior of all parallel nodes?
- When the total data size changes, is there any change to the workload of a task, and does it affect parallel behavior?

Assume we have n nodes, parallel behavior refers to whether n parallel tasks can by running simultaneously. It is in contrast with such a scenario: when the number of nodes reaches a certain value, because of some bottleneck, the growth of nodes couldn't give rise to the growth of parallel tasks. That is to say, parallelism is limited to a bottleneck.

The following subsection will give a detailed analysis on the influence of increasing computing nodes and total input data size.

4.2.1 The effect of increasing total data size

Assume we keep the number of nodes fixed, we analyze the effect of increasing total data size for map and reduce phase separately.

Considering map phase, the input data is a number of evenly partitioned data splits(only one split is an exception). This partition process is performed before any MapReduce program is running. Once we upload any data to Hadoop Distributed File System, data is automatically partitioned to multiple splits according to a certain split size. Because of this, the number of unique splits equals the number of map tasks and they form a queue waiting to be processed. If we keep the split size fixed, increasing the total data size will not affect the size and workload of such a working unit. It just increases the total number of map tasks. Further more, Hadoop MapReduce only allows us to load a certain number of map tasks into the memory of one node each time, eg. two map tasks. This threshold value makes sure memory and CPU are utilized regularly.

As for reduce phase, its input is constructed from many map tasks' intermediate results. These results are stored in local disks. because before map phase is finished the size of intermediate results is unknown, the number of reduce tasks is a user-defined value. That is to say Hadoop MapReduce does not automatically increase the number of reducers. If we increase this value proportionally while increasing total data size, the input data size for a reducer also doesn't change.

According to the analysis of above, if we keep the split size fixed and tune the number of reducers proportional to total data size, both the workload of a map and reduce task are not affected by increasing total data size.

4.2.2 The effect of increasing nodes

A direct effect of changing the number of nodes is the change to the arrangement of data splits. More nodes means more splits can be processed in parallel. Its negative effect is when the number of nodes reaches a certain value, increasing nodes couldn't bring about any speedup. Namely, it exposes the speedup bottleneck. This bottleneck might come from the problem itself, or the limitation of shared resources. As mentioned before, if a problem's non-parallelizable workload occupies 5% of the whole workload, the maximum speedup is 20 [1]. More nodes means more traffic and heavier burden to the network. In fact many resources can be seen as shared resources, for instance, when a node serves as an Http server for all the nodes in the network, then the node itself is a shared resource. Another type of shared resource is the network relay device. Such devices are hubs,switches routers and so on. They are shared by a set of nodes. We also took the same analyzing strategy as before: fix the total data size, just increase nodes.

Considering map phase, the only process might be affected is partition function. But partition function just partitions a map task's results to reducers and the number of them is not related to the number of parallel nodes, for example, partition function for WordCount program is $hash(key)\ mod\ R$, where R is the number of reducers, hence partition function is not affected. Further more there is no communication between any pair of map tasks. Hence, the whole map phase is not affected by the number of nodes.

Considering reduce phase, which has three subphases:copy,sort and reduce. Both sort and reduce operation are handled independently inside a node, thus from the perspective of parallel nodes, more nodes will surely accelerate sort and reduce operation. But from the perspective of an individual node, there is no impact on the processing time of sort and reduce operation. As for copy operation, it is involved with the utilization of shared resources such as the network and other nodes. On the other hand, increasing the nodes will evidently result in more reducers run in parallel. Thus, they will put heavier burden on the shared resources. This scenario is shown in Figure 4.2.

Figure 4.2 gives us hint that increasing nodes but not increasing the capacity of shared resources will give rise to speedup bottleneck to subphase copy. Figure 4.3 can help us understand what this bottleneck means. The lower part of the figure shows intermittent copy, which represents current MapReduce's actual workflow. If we extract all the copy

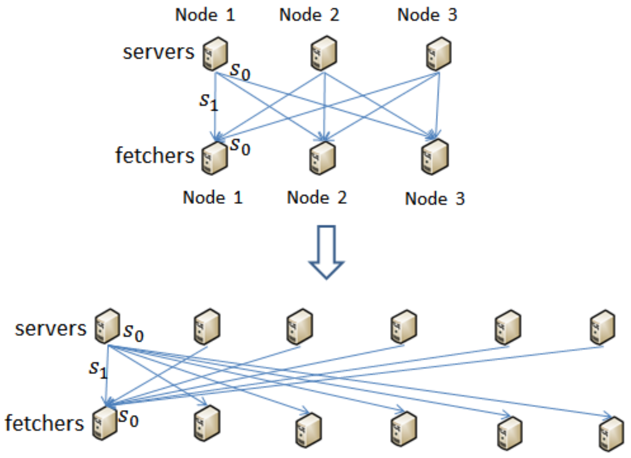


Figure 4.2: Scenario of expanding the network

operation from reduce phase and connect them together, we have the three-phase model shown in the upper part in Figure 4.3. We name this centralized copy as *once-and-for-all* copy. The meaning of bottleneck is that, the time consumption of copy phase shown in three-phase model couldn't be reduced even if we increase the number of nodes. Because of this, the speedup of copy phase couldn't grow. This speedup behavior is roughly shown in Figure 4.4. Even though copy speedup doesn't affect map speedup and other phases' speedup, as a part of the whole job, according to Equation 2.2, it does affect the final speedup.

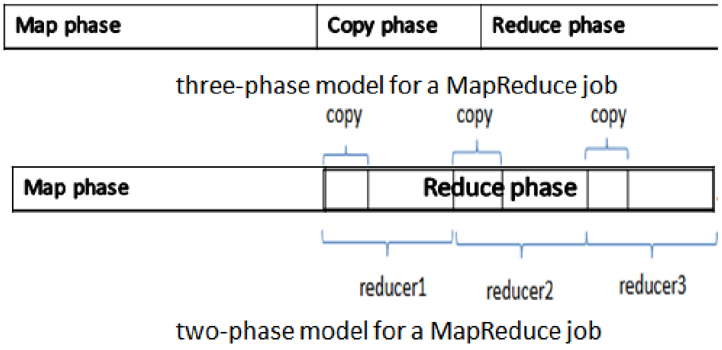


Figure 4.3: Models to describe a MapReduce job

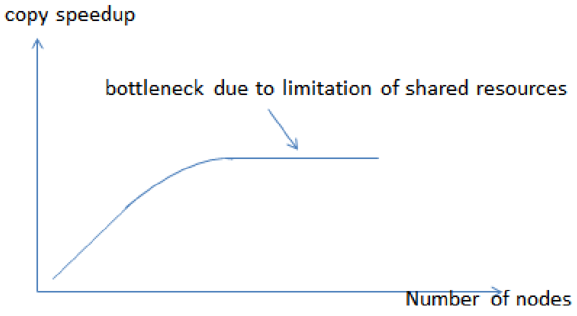


Figure 4.4: Bottleneck due to limitation on shared resources

In a word, two types of physical entities may cause speedup bottleneck for copy phase. Both a network relay device and any individual node may cause speedup bottleneck for copy phase.

4.2.2.1 Speedup bottleneck due to network relay device

As mentioned before, a reducer fetches data from the network sequentially, but a batch of reducers fetch data in a parallel manner. There is no doubt that data exchange must be based on the most important shared resource:network. Therefore, due to the limitation of network bandwidth, the speedup of data fetching couldn't grow all the time. For instance, if one reducer is supposed to fetch 100MB data from 10 parallel nodes, then 10 parallel reducers require to fetch 100*10=1000MB data simultaneously from the network. Thus 1000 nodes requires to fetch 1000*100=97.7GB of data simultaneously. It is obvious that transmitting 97.7GB of data lays much more burden on network than 1000MB. It is hard to guarantee that 97.7GB and 1000MB of data finish their transmission within the same amount of time.

The ideal situation of such data parallelism has the following characteristics:(1) one reducer fetches d bytes of data from a network consumes the same time with n parallel reducers fetching nd bytes of data; (2) if one node fetches data from the network with transmission speed s, k nodes should have transmission speed ks.

Considering a star network, all the nodes are connected to a centralized switch or other devices. The switch helps nodes achieve point-to-point communication. Normally, the switch has a certain processing capacity. We denote it as h (bytes per sec), which is also a constraint for data transmission. The h means that at most h bytes of data can be processed simultaneously by the switch every second. we consider such a scenario:

> Suppose we have a MapReduce job, which has D bytes of data to transmit after map phase. We use n nodes and $2n$ nodes to run the job separately. Denote $T_c(n)$ as its total data fetching time by use of n nodes. Accordingly, denote $T_c(2n)$ as the total data fetching time for $2n$ nodes. Denote $t_c(n)$ as one reducer's data fetching time from n nodes, accordingly $t_c(2n)$ is used for $2n$ nodes. Denote s_0 as point-to-point transmission speed. Therefore, n nodes require to have transmission speed ns_0 and $2n$ nodes require transmission speed $2ns_0$. Assume for both n nodes and $2n$ nodes, we use r reducers, therefore, each reducer is supposed to fetch data $d = \dfrac{D}{r}$ bytes. We ignore time delay due to

Http connection setup and teardown.

We analyze the following situations:

1. $ns_0 < 2ns_0 \leq h$. Since each reducer fetches d bytes of data sequentially, naturally $t_c(n) = t_c(2n) = \frac{d}{s_0}$. According to $D = T_c(2n) \cdot 2ns_0 = T_c(n) \cdot ns_0$, we have $T_c(2n) = \frac{1}{2}T_c(n)$. Under this situation, both n nodes and $2n$ nodes fetch data simultaneously.

2. $h \leq ns_0 < 2ns_0$. Under this situation, h causes speedup bottleneck. Following the same logic as the item above, we have $D = T_c(2n) \cdot h = T_c(n) \cdot h$, thus $T_c(2n) = T_c(n)$. Under this situation, even if we have n or $2n$ nodes, actually only $\frac{h}{s_0}$ nodes fetch data simultaneously or the transmission speed of each node decreases from s_0 to $\frac{h}{n}$ and $\frac{h}{2n}$. These two situations are accordingly manifested by: (1) $t_c(n) = t_c(2n)$; (2) $t_c(n) < t_c(2n)$.

The main difference between these two situations is $T_c(2n) = \frac{1}{2}T_c(n)$ becomes $T_c(2n) = T_c(n)$. Actually, this difference represents two different data fetching behaviors. They are described in Figure 4.5. We name the left data fetching behavior as *full copy*, the middle one as *partial copy* and the right one as *downgraded full copy*. Both partial copy and downgraded full copy indicate that data fetching bottleneck occurs.

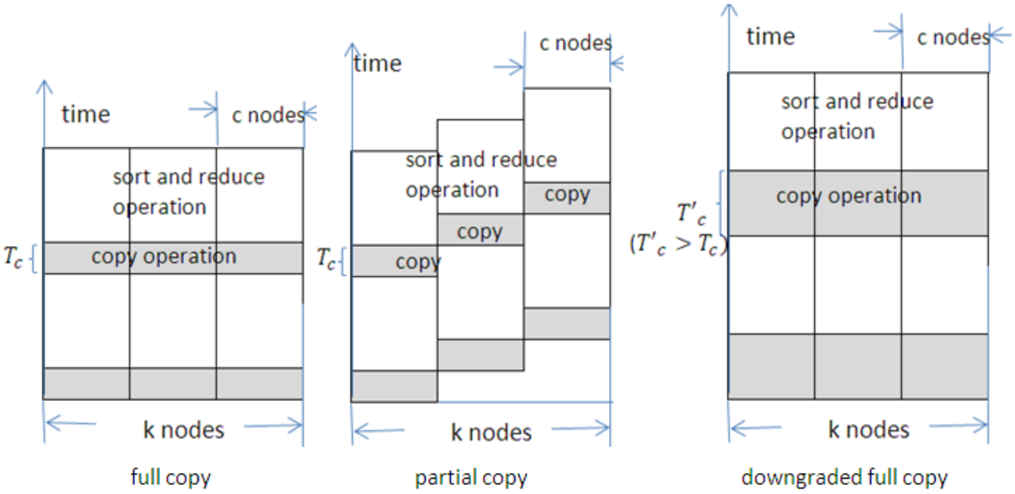


Figure 4.5: Three data fetching behaviors

Considering a two-level tree network, if two levels of switches have sufficient capacity, the network can be seen as a star network.The difference between transmission speed of in-rack nodes and cross-rack nodes makes two-level tree network a bit more complicated than a star network. We denote s_0 as in-rack point-to-point transmission speed, and s_1 as cross-rack point-to-point transmission speed. We denote $d = d_0 + d_1$, where d represents a reducer's input data and d_0 is the part of data distributed within the same rack with that reducer, and d_1 represents the data distributed outside the reducer's rack. Figure 4.6 shows these notations in detail. Obviously, s_0 is greater than s_1.

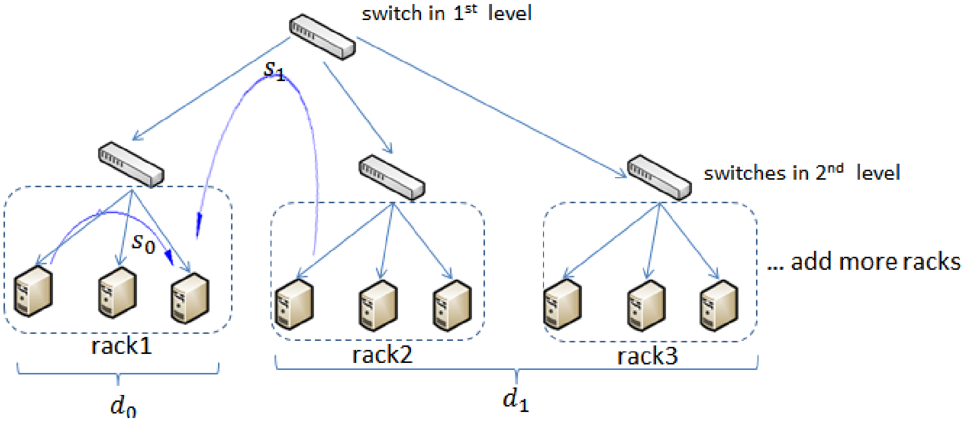


Figure 4.6: Expand two-level network

For a large cluster, expanding the network is mainly done through increasing the number of racks. Thus the bottleneck is often caused by the first-level switch, rather than the second-level switch. Similar to the analysis above, we have $t_c(n) = \frac{d_0}{s_0} + \frac{d_1}{s_1}$. Without considering any bottleneck, the two-level tree network can actually be approximate to a star network, therefore $T_c(2n) = \frac{1}{2}T_c(n)$.

Under the premise of $d = d_0 + d_1$, we have $t_c(n) = \frac{d_0}{s_0} + \frac{d_1}{s_1}$, where d is fixed all the time. The increase of n or rather the increase of racks will increase d_1 and decrease d_0. Since $s_0 > s_1$, the increase of $\frac{d_1}{s_1}$ is greater than the decrease of $\frac{d_0}{s_0}$. As a result, $t_c(n)$ will increase when we increase n, even if there is no bottleneck.

4.2.2.2 Speedup bottleneck due to an Http server's capacity

The capacity of an Http server can cause speedup bottleneck. In Hadoop MapReduce , every node serves as a server. An Http server's capacity depends on a server's memory and CPU capacity. Denote this maximum Http connections as c. Similar to the analysis above we list the following situations:

1. $c \geq 2n$. This situation is similar to the first situation of our previous analysis on the bottleneck caused by network relay device . It is not difficult to get $t_c(2n) = t_c(n)$

and $T_c(2n) = \frac{1}{2}T_c(n)$. Both n nodes and $2n$ nodes fetch data simultaneously

2. $c \leq n$. Under this situation, n nodes are divided into $[\frac{n}{c}] + 1$ groups. The operation $[\frac{n}{c}]$ means calculating integer part of $\frac{n}{c}$. Data transmission is handled sequentially among these groups. Under this situation, only c nodes can fetch data simultaneously. Therefore, $t_c(n) = t_c(2n)$ and $T_c(n) = T_c(2n)$.

Obviously, the situation of $c \leq n$ causes speedup bottleneck. even though we have n or $2n$ parallel nodes, it only allows c nodes fetch data in parallel.

4.2.2.3 Decide boundary value

Our previous analysis shows that once we meet any bottleneck, full copy will either turn to partial copy or downgraded full copy. It is important to find out when this change happens. We denote b as the boundary value to describe this copy behavior change. When the number of nodes is less than b, the copy behavior is full copy, when it surpasses b, it turns to be either partial copy or downgraded full copy. As for which exact copy behavior it turns to be, it depends on how the infrastructure program configure the switch. We can use an empty MapReduce program to detect whether our customized number of nodes is within boundary b. We design such a MapReduce program that the map operation does nothing but transmits data directly to reduce phase, and the reduce operation also does nothing. We illustrate this experiment by use of Figure 4.7.

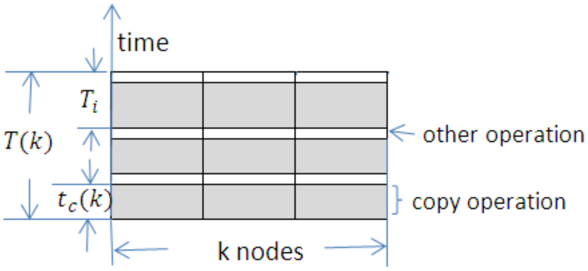


Figure 4.7: Boundary detection experiment illustration

Figure 4.7 shows that k is less than boundary b. $T(k)$ represents the execution time of the whole MapReduce job by use of k nodes; T_i stands for the ith reducer's execution time and $t_c(k)$ represents average copy time of a reducer. We do an experiment by use of k nodes at first. Then we increase k to be k'. We aim at detecting whether k' nodes exceeds boundary b. In both experiments, the input data is the same. Further more, the number of mappers and reducers are kept the same. The judgement steps are described as follows:

1. Firstly, compare $T(k)$ with $T(k')$ first, if $T(k) = T(k')$ then the copy behavior of k' nodes is full copy, otherwise it is either downgraded full copy or partial copy.
2. If $T(k) \neq T(k')$, compare $t_c(k)$ with $t_c(k')$. If $t_c(k) = t_c(k')$, the copy behavior is partial copy, otherwise it is downgraded full copy.

Actually, this experiment is not every efficient, because it requires to test the same set of data on both k and k' nodes. Once k is far less than k', the experiment on k nodes will still consume a large amount of time. But this experiment does give us hint to propose another more efficient experiment. This experiment scheme is described as follows:

> Make an empty MapReduce program as described before. Test small amount of data by use of k nodes and compute $t_c(k)$. Test proper amount of data on k' nodes. This data is larger than previous data on k nodes. Accumulate all reducers' execution time in the experiment of k' nodes as T. The ith reducer's execution time is marked as T_i in Figure 4.7. The judgement steps are described as follows:
>
> 1. Compare $\frac{T}{k'}$ with $T(k')$, where $T(k')$ is the execution time for the whole MapReduce job. If $\frac{T}{k'} = T(k')$, then the copy behavior is either full copy or downgraded full copy.
> 2. If $\frac{T}{k'} = T(k')$, compare $t_c(k)$ with $t_c(k')$, if $t_c(k) = t_c(k')$ then the copy behavior is full copy, otherwise it is downgraded full copy.

4.2.2.4 Summary

We mainly analyzed the effect of increasing nodes in a star network. Its negative effect is full copy may become partial copy and downgraded full copy. We suppose partial copy and downgraded full copy are two forms of bottleneck. The conditions of full copy are listed as follows:

1. $ns_0 < h$, where h represents capacity of central switch and s_0 represents point-to-point transmission speed.
2. $c \geq n$, where c represents the maximum Http connections to a server and n represents the number of servers.

These two conditions are manifested in two aspects:

1. One reducer's fetching time $t_c(n)$ is fixed, even if n grows.
2. When n grows, the data fetching time of a parallel group $T_c(n)$ will decrease.

In fact, those two conditions essentially stand for the same thing: the limitation on shared resources. A server node has connections with all other nodes. It means it is consumed or shared by all other nodes. Its Network Interface Card(NIC), CPU and memory are shared resources. Data exchange must make use of the network, thus the replay devices on network become shared resources. As we increase nodes, the limitation on any shared resources can bring about bottleneck to subphase copy. Once shared resources meet their limitation, copy cannot speed up, but other phases can still speed up. Even if we meet any form of bottleneck, previous experience is still useful.

Our analysis ignored time delay caused by Http connection setup and teardown. Considering this effect, one reducer's data fetching time $t_c(n)$ is not always fixed. As the increase of n, $t_c(n)$ will have slight increase. We also did a short analysis on a two-level tree network. From a certain angle, it can be seen as a star network. The increase of n will also cause slight increase of $t_c(n)$ even if no bottleneck occurs. However, within a certain allowable range, we can ignore this slight increase.

The conclusion that one reducer's fetching time $t_c(n)$ is fixed is of great significance. It leads to the conclusion that one reducer's processing time T_r is fixed. This is because a reducer is composed of three operations: copy, sort and reduce and none of their time consumption is affected by n. Considering one mapper's processing time T_m is also fixed, T_m and T_r can be seen as useful learning experience. That is to say, they can be used to estimate the execution time of a MapReduce job, which aims at handling a larger scale of data on larger scale of nodes.

4.3 Wave model

Our previous analysis gives us some hint to propose a computation model to estimate execution time for a MapReduce job. In order to facilitate our description we define a term *wave*.

Definition: A *wave* is a group of parallel tasks, the length of which is the number of parallel nodes n. There is map wave in map phase and reduce wave in reduce phase. The number of waves is computed by:

$$N_{wave} = [\frac{N_{task}}{n}] + 1$$

where N_{task} represents the number of tasks. The operation $[\frac{N_{task}}{n}]$ means obtaining the integer part of $\frac{N_{task}}{n}$. The number of waves N_{wave} indicates that roughly N_{wave} tasks will be allocated to one node. Figure 4.8 shows that three tasks form a wave, which are represented by three vertical grids in the figure. Further more, in order to facilitate our description, we name the series of tasks executed by an individual node as a task chain. Figure 4.8 shows that three nodes have three task chain.

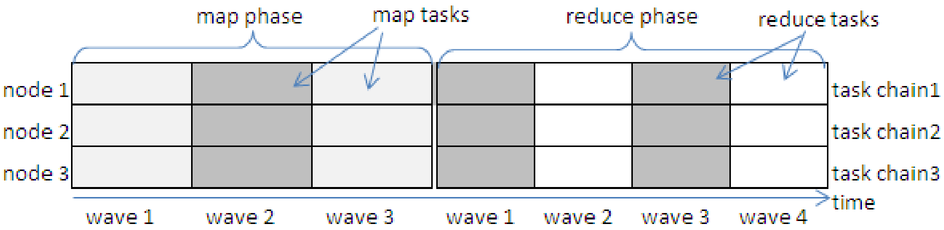


Figure 4.8: The illustration of concept “wave”

4.3.1 General form

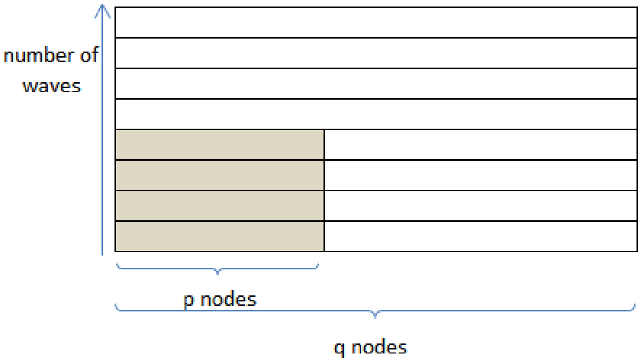


Figure 4.9: General form of wave model

This model is described in Figure 4.9. This figure shows that we apply the learning experience on p nodes to q nodes($p < q$). The most important condition for this model is q is less or equal than boundary b. How to test whether q is within boundary b has been illustrated in subsection 4.2.2.3 Decide boundary value.

This model derives the following expression to estimate execution time:

$$\begin{aligned}\hat{T} &= \hat{T_M} + \hat{T_R} && (4.2)\\ &= T_m N_{mw} + T_r N_{rw} && (4.3)\end{aligned}$$

where N_{mw} is computed by $[\frac{N_m}{n}] + 1$ and N_{rw} is computed by $[\frac{N_r}{n}] + 1$.

The meanings of notations are described in Table 4.1:

$\hat{T}$	estimation of execution time
$\hat{T_M}$	the estimated time consumption for map phase
$\hat{T_R}$	the estimated time consumption for reduce phase
T_m	the time consumption of a map task
T_r	the time consumption of a reduce task
N_{mw}	the number of map waves
N_{rw}	the number of reduce waves
n	the number of parallel nodes
N_m	the number of map tasks
N_r	the number of reduce tasks

Table 4.1: Notations for general form of wave model

Assume we have verified q is within the boundary b. The steps of how to accumulate learning experience and how to apply the learning experience on p nodes to q nodes are described as follows:

1. Do training experiment on p nodes by use of small sale of data, obtain T_m and T_r.

2. When it comes to large scale of data, according to the total input size and split size set in HDFS, compute the number of map tasks N_m and the number of reduce tasks N_r.

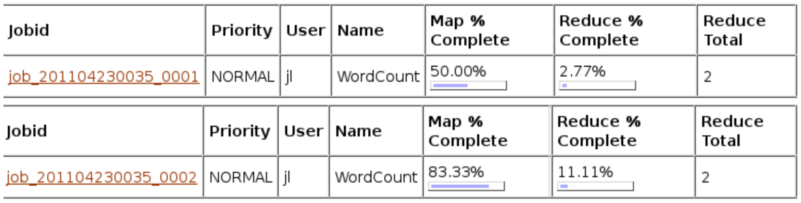
3. Compute the number of map waves N_{mw} and the number of reduce waves N_{rw} according to $N_{wave} = [\frac{N_{task}}{n}] + 1$.

4. Use T_m and T_r obtained in training experiment, N_m and N_r to compute $\hat{T} = T_m N_{mw} + T_r N_{rw}$. $\hat{T}$ is our estimated value for large scale of data on q nodes.

From the step description above, we see that our useful experience information is T_m and T_r. From now on we will not provide detailed step description unless it's necessary. Pointing out what is experience information is enough for readers to catch our ideas.

4.3.2 Slowstart form

The model shows in Figure 4.9 just presents the ideal situation that reduce phase waits for the completion of all mappers. But Hadoop MapReduce has a *slowstart* mechanism, which needs special treatment. Slowstart value indicates that a part of reducers can be launched after the a fraction of mappers are completed[26]. Comparing with reducers wait for the completion of all mappers, this slowstart mechanism appears to be more efficient. Figure 4.10 shows the effect of slow start. As long as some map tasks are finished, their results can be copied to reducers. The whole copy phase for a reduce task occupies 33.3% [1] progress in that reduce task.



Jobid	Priority	User	Name	Map % Complete	Reduce % Complete	Reduce Total
job_201104230035_0001	NORMAL	jl	WordCount	50.00%	2.77%	2

Jobid	Priority	User	Name	Map % Complete	Reduce % Complete	Reduce Total
job_201104230035_0002	NORMAL	jl	WordCount	83.33%	11.11%	2

Figure 4.10: Slowstart

Even though Hadoop MapReduce utilizes slowstart to warm up reduce tasks, only the first wave of reduce tasks are affected. Further more, slowstart only performs copy operation earlier. This is because sort and reduce operation couldn't start until the data is completely copied from all mappers. Reducers have to start sort and reduce operation after the arrival of the last (key,value) pair from the last mapper. Therefore, although the first wave of reduce tasks starts early, they are actually in an "idle" state, which means they occupy computation resources by fetching data, but couldn't start sort and reduce operation. The math computation process of slowstart form is illustrated under the help of Figure 4.11.

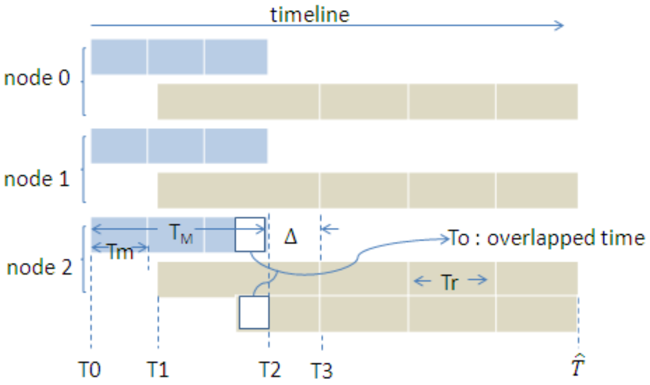


Figure 4.11: Slowstart analysis model

A part of the notations can be found in Table 4.1, and the same notation stands for the same meaning. New notations are described in the following Table 4.2:

[1] This value is obtained from Hadoop MapReduce runtime monitoring webpage.

$\hat{T}$	the estimation of completion time
$T0$	the start time of job
$T1$	the start time of reduce phase
$T2$	the end time of map phase
$T3$	the end time of first reduce wave
Δ	equals $T3 - T2$
T_{ov}	overlapped time,equals $T_r - \Delta$

Table 4.2: Notations for slowstart form of wave model

The goal here is to compute $\hat{T}$. Estimating T_M is easy, which is achieved by $\hat{T_M} = N_{mw}T_m$, however because of the abnormal behavior of the first reduce wave, $\hat{T_R}$ couldn't be simply computed by $N_{rw}T_r$.

Let $T'_R = N_{rw}T_r$, refer to Figure 4.11, we have:

$$\begin{aligned}\hat{T} &= T_M + T'_R - T_{ov} && (4.4)\\ &= T_m N_{mw} + T_r N_{rw} - (T_r - \Delta) && (4.5)\\ &= T_m N_{mw} + T_r N_{rw} - (T_r - T3 + T2) && (4.6)\end{aligned}$$

An important question here is as long as T_m and T_r are fixed, Δ doesn't change even if scaling plan changes. As we mentioned before, reduce phase couldn't start sorting and reduce until map phase is completely finished, therefore it is reasonable to treat T_{ov} as the time consumption for copy operation of the first reduce wave. We see from the expression above that our experience information is T_m, T_r and T_{ov}.

Figure 4.11 shows that slowstart only saves time T_{ov}, which is less than T_r. Considering huge amount of reduce tasks, this amount of time is ignorable. Therefore, the following models will not consider slowstart situation, even if Hadoop MapReduce has provided such a mechanism.

4.3.3 Speedup bottleneck form

We analyzed several possible situations which might cause speedup bottleneck. Because speedup bottleneck just happens in subphase copy, if we treat it differently, we can still find proper models to estimate execution time. According to Equation 2.2, as long as subphase copy doesn't occupy a great percentage in the whole job, it is still meaningful to intensify parallelism for other phases by increasing nodes.

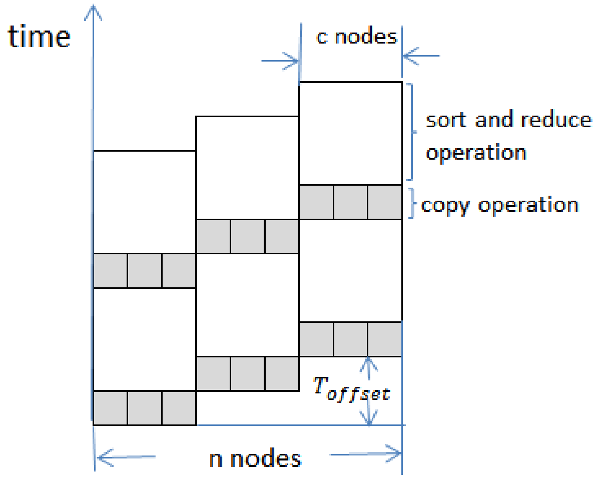


Figure 4.12: Partial copy

Our previous analysis has revealed two copy behaviors caused by copy bottleneck. These two copy behaviors are partial copy and downgraded full copy. By use of Figure 4.12, we can use the following expression to estimate execution time:

$$\hat{T} = \hat{T_M} + \hat{T_R} + T_{offset} \quad (4.7)$$

$$= T_m N_{mw} + T_r N_{rw} + T_{offset} \quad (4.8)$$

where T_m, T_r and T_{offset} are experience information.

Considering downgraded full copy, we concluded that one reducer's data fetching time $t_c(n)$ will increase if n increases. Figure 4.13 described this phenomenon clearly. Actually, even if we don't consider this bottleneck, considering time delay brought by Http connection setup and teardown will also have some slight impact on $t_c(n)$. Further more, our analysis on two-level tree network also revealed that the increase of n will increase $t_c(n)$. This side-effect of increasing n is not so severe as the effect of bottleneck, but they behave very similarly. Thus, we don't aim to distinguish them here.

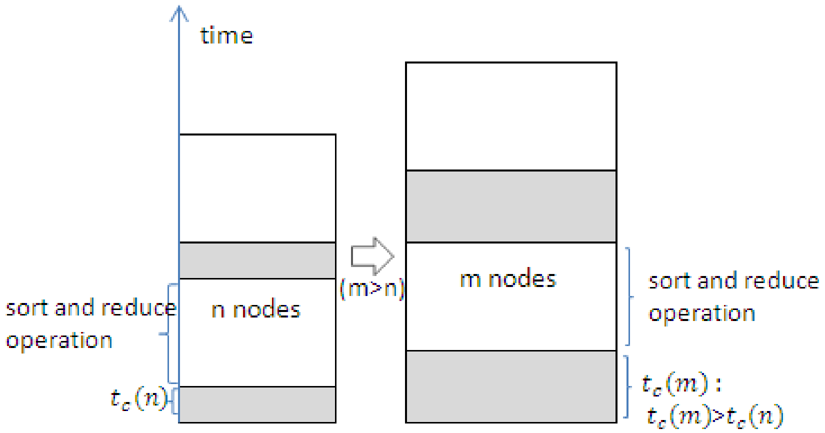


Figure 4.13: Downgraded full copy

On the basis of the empty MapReduce program we proposed to test whether it is full copy or downgraded full copy in Subsection 4.2.2.3 Decide boundary value, we propose a solution to estimate execution time when we meet $t_c(m) > t_c(n)$ where $m > n$.

Assume we aim to estimate the execution time of a practical problem on m nodes. We use three experiments to achieve our goal:(1) empty MapReduce experiment on n nodes where n is far less than m; (2) empty MapReduce experiment on m nodes; (3) practical problem-solving experiment on n nodes. From Figure 4.13 we see that the execution time on sort and reduce operation do not change, no matter we use n or m nodes. Therefore, our main goal is to estimate the copy time of practical experiment on m nodes, which is denoted as $t'_c(m)$. Similarly, we denote the copy time of practical experiment on n nodes as $t'_c(n)$. We estimate $t'_c(m)$ by the following expression:

$$t'_c(m) = t'_c(n) * \frac{t_c(m)}{t_c(n)}$$

According to the execution workflow of a reduce task. Its execution time T_r is composed of three parts:T_{copy}, T_{sort} and T_{reduce}. Comparing the experiment on n and m nodes, only the value of T_{copy} is changed from $t'_c(n)$ to $t'_c(m)$. Therefore, we just replace $t'_c(n)$ with $t'_c(m)$ to compute the execution time of a reduce task and finally compute the whole execution time by $\hat{T} = T_m N_{mw} + T_r N_{rw}$ (Expression 4.3).

4.3.4 Model verification for general form

We mainly did experiments on WordCount and Terasort program. We have two types of experiments:traning experiments and test experiments. Training experiments are used to accumulate experience, while test experiments are used to verify the correctness of our model. We obtained the start time and end time of every task. The duration is represented by their gap. We gathered them to form task chains and traced their execution time.

In stead of executing a job several times to compute its average execution time, we measure it by computing the expectancy from task chains. The advantage of this is we can reduce the number of experiment time.

We first present the behavior of map and reduce wave. Figure 4.14 and 4.15 show map and reduce wave separately. This result shows the experiment of 4 nodes processing 1.86GB data. For WordCount and Terasort program, we customized five experiments for each program. The verification results are shown in Table 4.3 and Table 4.4.

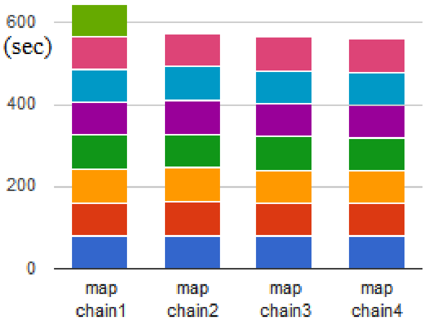


Figure 4.14: Map wave of WordCount.

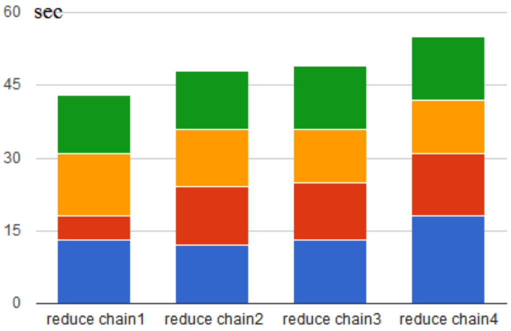


Figure 4.15: Reduce wave of WordCount.

	4nodes			8nodes			16nodes		
	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap
929MB	learn	learn	-	185.375	172.26	13.115	-	-	-
1.86G	702.25	689.4	12.85	360.75	344.52	16.23	-	-	-
1.86G*2	-	-	-	-	-	-	363.63	344.52	19.11

Table 4.3: Verify wave model for WordCount program

	4nodes			8nodes			16nodes		
	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap
2GB	learn	learn	-	161	156.12	4.88	-	-	-
4GB	657.5	624.48	33.02	313	312.24	0.76	-	-	-
8GB	-	-	-	-	-	-	318.9	312.24	6.66

Table 4.4: Verify wave model for Terasort program

These two tables show that the gaps are within 10% of $min\{T_{measure}, \hat{T}\}$. It means our measured values are consistent with estimated values. Theoretically speaking, if T_m and T_r in our learning experiment are closer to T_m and T_r in new experiments, the gap should be smaller. However, the fact is T_m and T_r are just average of a set of values. It is the fluctuation or deviation of this set of values decides whether estimated value are closer to measured value. From this point of view, we see that wave model requires a strict homogeneous computing environment. But on the other hand, if accuracy requirement is not very high, the simplicity of wave model is a good reason of choosing it.

4.3.5 Model verification for slowstart form

The verification for slowstart form is shown in Table 4.5. This experiment result verifies that our computation method for slowstart situation is reasonable. The gaps are still within 10% of $min\{T_{measure}, \hat{T}\}$. Even though slowstart doesn't really save too much time, it does introduce an interesting problem: the current slowstart just serves the first reduce wave, why designers don't let it serve all the reduce waves is an interesting problem. Logically speaking, after a map task is finished, which part of its result belongs to which reducer is fixed by partition function, then the system knows which result should be delivered to which target machine.

	8 nodes(Terasort)			8 nodes(WordCount)		
	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap
2 GB	-	-	-	-	-	-
4 GB	384.375	382.956	1.419	548.05	538	10.05

Table 4.5: Verify wave model for slowstart form

Essentially, this problem is about the argument between intermittent data exchange and once-and-for-all data exchange. We noticed that in previous version Hadoop pre-0.18.0, once-and-for-all data exchange was used. Yahoo's One Terabyte Sort experiment shows this clearly. The task execution details is shown in Figure 4.16[23].

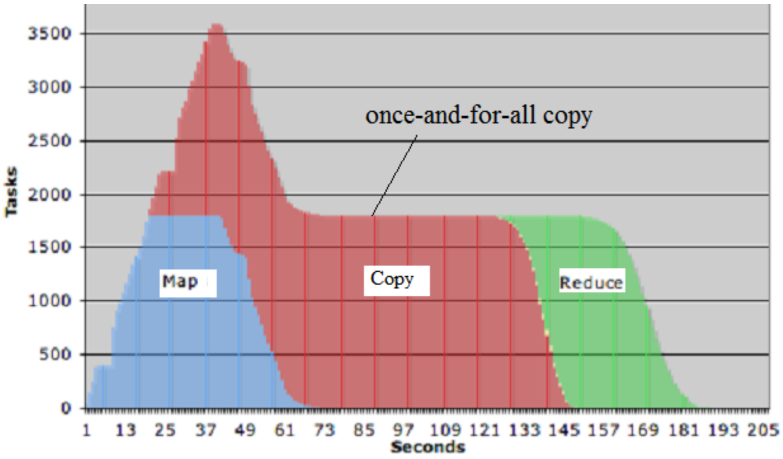


Figure 4.16: Once-and-for-all copy[23]

From our point of view,once-and-for-all copy is a bandwidth intensive process. Data exchange is straightforward, but it also shows that CPU and memory are not fully utilized in this process. Thus, recent Hadoop versions(our experiment version is 0.20.2) use intermittent copy to replace it. For intermittent copy if a part of reducers take less copy time than others,they can start sort and reducer operation earlier, thus less parallel nodes are using the network. That is to say not all the nodes utilize network simultaneously. This alternate utilization of network relieves the burden on network. Figure 4.17 shows slowstart copy pattern used in Hadoop 0.20.2.

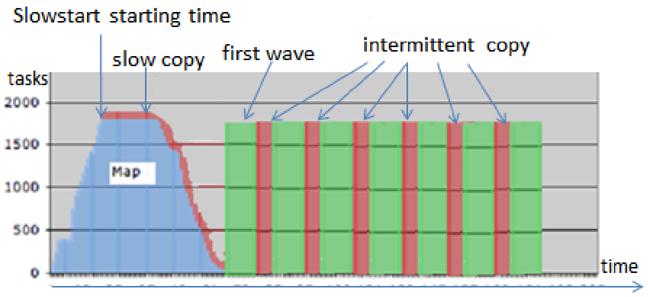


Figure 4.17: Current slowstart copy pattern

The problem with this slowstart copy pattern is during map phase copy operation just serves the first wave, comparing with once-and-for-all copy it is quite slow. Combining intensive copy in previous pattern with current intermittent copy, we propose a new copy pattern. This copy pattern is shown in the Figure 4.18. This copy pattern combines the benefits of intermittent copy and intensive copy.

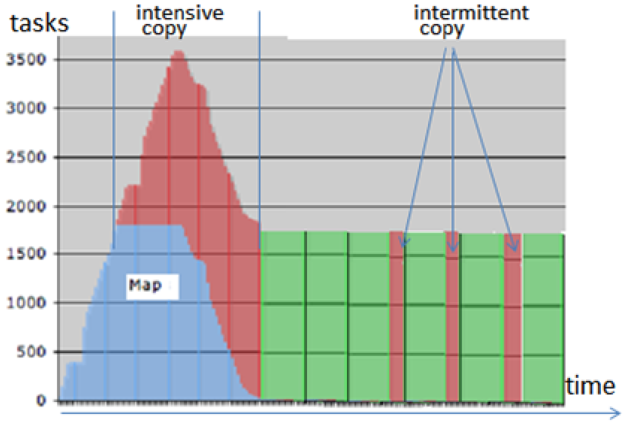


Figure 4.18: Proposed new copy pattern

4.4 Sawtooth model

The previous section described wave model, which focuses on the group behavior of all parallel nodes. In fact, we can also focus on the individual behavior of every node. Based on this idea, we come up with a *Sawtooth model*. We suppose that slowstart just reduce the processing time for first reduce wave. We do not provide the slowstart form in this section.

4.4.1 General form

This model is described in Figure 4.19.

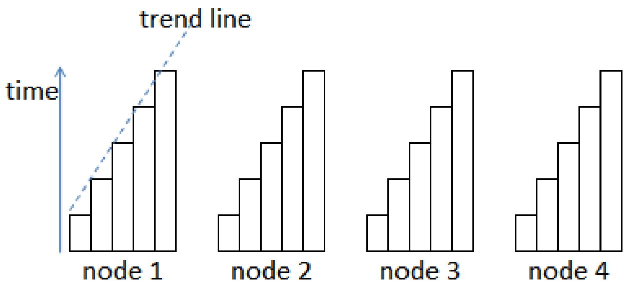


Figure 4.19: Sawtooth model

Basically, from each machine's view, as tasks are gradually processed by itself, their execution time forms a trend line. Therefore, we can use some math tools such as linear regression to generate its trend function and finally estimate the whole execution time for a job by collecting each machine's trend function. An important thing is this model should be applied to map and reduce phase separately. We add their estimation value together to be the execution time for a job. In order to make our results more accurate, it is better to accomplish more waves to draw trend line.

The details of our ideas are summarized to be the following steps, we just use map phase as our illustration example:

1. Accumulate experience by generating linear function for every experiment node: $f_i(x) = a_i x + b_i$, where $1 \leq i \leq n$, n is the number of our experiment nodes. The gradient a_i represents the average execution time of a map task on $node_i$.

2. Facing with large scale of data, estimate the number of waves N_{mw}. Assume we have increased computing nodes from n to m;

3. Compute $f_i(N_{mw})$, where $1 \leq i \leq n$;

4. Sort $f_i(N_{mw})$, and balance them. This balanced value is our estimated value. The reason for this step is there might be a big gap between the biggest $f_j(N_{mw})$ and the smallest $f_k(N_{mw})$. If this gap exceeds execution time of a map task a_j and a_k, then we should amend this by deducting one task from $node_j$ and adding one task to $node_k$. This goal is achieved by computing $f_j(N_{mw} - 1)$ and $f_k(N_{mw} + 1)$. This procedure is performed recursively until the gap is small enough.

The estimation for reduce phase is the same as map phase. After this is done, we add these two estimated value together to be our final estimation.

4.4.2 Model verification for general form

We first present the behavior of map and reduce sawtooth. Figure 4.20 and 4.21 show map and reduce sawtooth separately. This result shows the experiment of 4 nodes processing 1.86GB data. These two figures match our model very well.

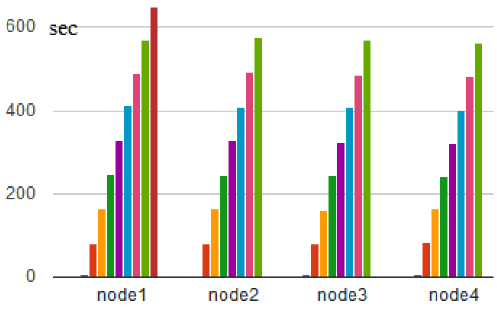


Figure 4.20: Map sawtooth of WordCount.

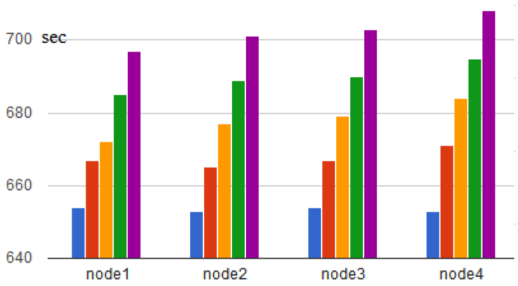


Figure 4.21: Reduce sawtooth of WordCount.

The verification of our computation method is shown in the Table 4.7 and 4.6. The results show that the gap is within 10% of $min\{T_{measure}, \hat{T}\}$. It means our measured values are consistent with estimated values.

	4nodes			8nodes			16nodes		
	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap
929MB	-	-	-	185.375	173.935	gap	-	-	-
1.86G	learn	learn	-	360.75	371.42	10.67	-	-	-
1.86G*2	-	-	-	-	-	-	363.63	371.42	7.79

Table 4.6: Verify sawtooth model for WordCount program

	4nodes			8nodes			16nodes		
	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap	$T_{measure}$	$\hat{T}$	gap
2GB	-	-	-	161	161.6	0.6	-	-	-
4GB	learn	learn	-	313	323.26	10.26	-	-	-
8GB	-	-	-	-		-	318.9	323.26	4.36

Table 4.7: Verify sawtooth model for Terasort program

4.5 Summary

This chapter is the most important chapter of all. We started from describing key assumptions and conditions. The proposition that execution time follows regular trend is the basis of our proposed models. The analysis on the effect of increasing total data size and increasing nodes revealed possible bottlenecks to copy operation. We name the two possible bottleneck behaviors as *partial copy* and *downgraded full copy*. Then, based on this analysis result we proposed two models to estimate execution time of a MapReduce job. These two models are named as *wave model* and *sawtooth model*. They mainly address two problems:(1) what experience information should be learned from training experiment; (2) What formula or expression should be used to compute the estimated value. In sawtooth model, we just provided the general form. Under the guidance of slowstart form and speedup bottleneck form of wave model, it is not difficult to adapt current sawtooth model to these situations. Due to budget limitation, we just provided possible bottleneck behaviors and conditions but didn't do practical experiment to verify them.

Chapter 5

Discussion

5.1 Scalability analysis

For a parallel system, scalability analysis is a necessary step. Scalability often covers two aspects: (1) scaling input data (2) scaling computing nodes. Scaling computing nodes is often analyzed via speedup, which is an important performance evaluation criteria. It aims at studying how much faster can a system achieves after expanding the number of nodes. In a multi-phase process, Equation 2.2 can facilitate our analysis quite a lot. In this aspect, MapReduce divides its process to be a map phase and reduce phase, which simplified our analysis. Even if when facing with a job chain, the speedup for the whole process can be computed easily by analyzing every job separately. Consider map phase individually, its scaling behavior is linear scaling, if no bottleneck occurs reduce phase also scales linearly. After combining them together, the scaling behavior is also linear scaling, which means speedup is linear.

Considering the effect of scaling data size on execution time, the ideal situation is increasing the data size just linearly increases its workload. In such case, the time consumption on $(n\ nodes, d\ megabyte)$ is the same with $(2n\ nodes, 2d\ megabyte)$. However, for many algorithms, workload is not proportional to their data size. For instance, considering bubble sort[4], whose average time complexity is $O(n^2)$. Increasing data size n to be $2n$ results in increasing workload approximately $\frac{(2n)^2}{n^2} = 4$ times. Therefore, assuming it's parallelizable, simply doubling computing nodes while doubling data size will not keep the execution time the same. Considering MapReduce programming model, when taking a task as granularity unit, linear scaling on total data size also brings about linear scaling on total workloads. However, one thing we should notice is the scaling of total data size can automatically scale the number of mappers, but it doesn't scale the number of reducers accordingly. Therefore in order to make the learning experience of one experiment consistent with another experiment, it is necessary to make sure the execution time of each reduce task is the same in both of the experiments. An easy strategy is to make the number of reducers proportional to the total data size. If we use 6 reducers for 1GB's data(the input size for map), then we use 12 reducers for 2GB's data.

	4nodes	8nodes	16nodes
929M	368.5	185.375	-
1.86G	702.25	360.75	-
1.86G*2	-	-	363.63

Table 5.1: Execution time for WordCount program(unit:sec)

	4nodes	8nodes	16nodes
2GB	318.5	161	-
4GB	657.5	313	-
8GB	-	-	318.9

Table 5.2: Execution time for Terasort program(unit:sec)

Due to the analysis above we conclude that by use of Hadoop MapReduce if workload is balanced and no bottleneck affects parallelism, execution time is proportional to data size and inverse proportional to the number of computing nodes. We describe this conclusion by the following expression:

$$T(nodes, datasize) \propto \frac{datasize}{nodes}$$

The experiment results in Table 5.1 and 5.2 are consistent with this expression. On the other hand, assume $T(1)$ is execution time on a single node, we have:

$$speedup = \frac{T(1)}{T(nodes, size)} \propto \frac{nodes}{datasize}$$

This expression tells that once we fix datasize, we can achieve linear scaling under certain conditions.

5.2 The essence of wave model and sawtooth model

In Chapter 2 we mentioned that our main method is to find out a function $T(nodes, datasize)$ and its valid boundary to estimate the execution time of a larger problem. Readers may wonder why we turned to wave model and sawtooth model afterwards. Actually, they stand for the same idea.

The key factor in wave model is the number of map waves N_{mw} and reduce waves N_{rw}. N_{mw} and N_{rw} are decided by both the number of nodes and total data size. In other words, instead of study $T(nodes, datasize)$ directly, we use a middle ware $N_{mw}(nodes, datasize)$ and $N_{rw}(nodes, datasize)$ to study it. We have a more simplified understanding: assume one node needs t time units to finish a MapReduce job, then n nodes need $\frac{t}{n}$ time units. It's like a string with length t is folded n times, then its folded length becomes $\frac{t}{n}$. The strategy of wave model is to compute the unit length of a string T_m, and then multiply it with the total number of units N_m and finally divide it by the number of parallel strings n. That is to say the folded length is expressed by $T_m N_m \frac{1}{n}$, where $\frac{N_m}{n}$ equals the number of waves.

The idea behind sawtooth model is similar to wave model. Sawtooth model generates trend line function $f_i(x) = a_i x + b_i$, and finally merge the value of every $f_i(x)$. The number of waves is used to replace x. Wave model computes the average value for a task unit first and then multiply it by number of waves N_{wave}. This multiplication is similar to replacing x with N_{wave}. Further more, in the expression $f_i(x) = a_i x + b_i$, the gradient a_i actually means the execution time of a task unit. If every trend function $f_i(x) = a_i x + b_i$ are the same, this particular case of sawtooth model can be seen as wave model. That is to say, wave model is actually a special case of sawtooth model.

The focus of wave model is the group behavior of all parallel nodes, but the focus of sawtooth model is every individual node. Therefore, wave model requires a heterogeneous computing environment more than sawtooth model. But on the other hand, wave model is obviously easier to perform. When the requirement of estimation accuracy is not very strict, the simplicity of wave model is a good reason for us to choose it.

5.3 Extend models to more situations

Even though our proposed models must satisfy certain conditions, in fact they can be extended to more situations. In some cases, small modification can adapt them to new situations. We list these possible situations as follows:

Workload of task unit changes periodically In this case, average time consumption for map task T_m and reduce task T_r also changes periodically. Wave model can be slightly modified to use a periodic function to describe. This scenario is described by the following figure:

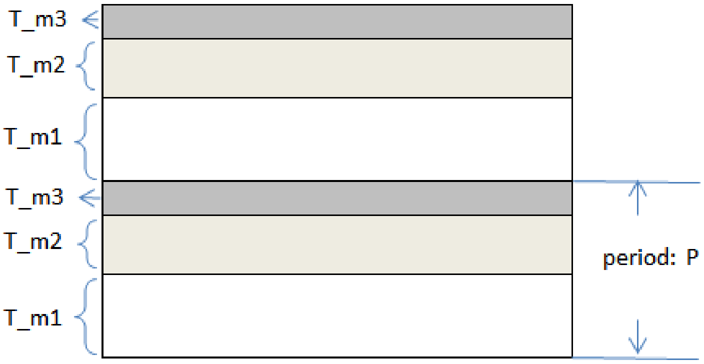


Figure 5.1: Periodical workload

Heterogeneous computing environment We assume workload density is balanced for the total data, but computing nodes have computation capacity variation or some nodes have better throughput capacity. In such case, sawtooth model can still be useful.

5.4 Deviation analysis

Like any experiment, when comparing measured value and estimated value, there is a gap. Why there is such a gap and how it comes out are important for scientific researchers. We measure the execution time of a job by all task chains' average ending time. But the estimated value is the addition of $\hat{T}_{Map}$ and $\hat{T}_{Red}$. In fact, there is a gap between $\hat{T}_{Map}$ and the start time of reduce phase, because reduce phase couldn't start until the last map task is finished. This scenario is described by Figure 5.2. This figure shows that even we use measured values T_{Map} and T_{Red} to compute execution time T, there is still a *gap*. Therefore, using estimated value $\hat{T}_{Map}$ and $\hat{T}_{Red}$, this gap should be wider.

Another reason of deviation comes from the computation problem. For example, we observed that WordCount is a CPU intensive program. When map phase is running, CPU utilization is always 100%, therefore, the worse the computers' quality is and the longer the map phase lasts the wider the gap is.

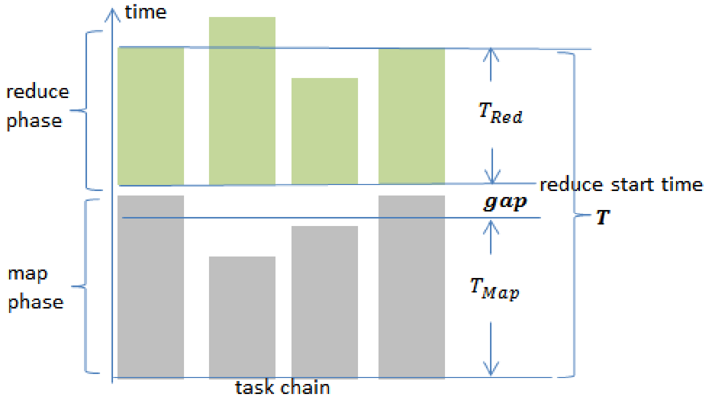


Figure 5.2: A deviation situation.

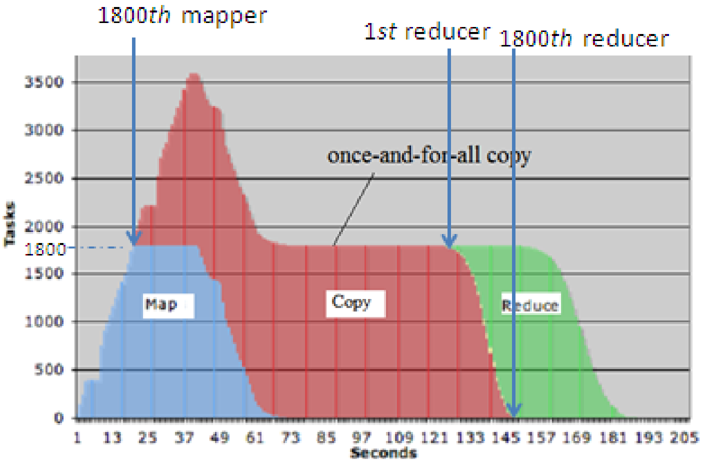


Figure 5.3: Not all tasks are launched at the same time[23]

.

For a large amount of parallel tasks for instance 1024 mappers, they cannot be launched completely at the same time. From Yahoo's experiment result of One Terabyte Sort showed in Figure 5.3 we see that approximately 1800 mappers are supposed to run in parallel. This figure roughly shows that the gap between the start time of the first mapper and $1800th$ mapper is within 25 seconds. Accordingly, the gap between the first reducer and the $1800th$ is also within 25 seconds. This fact tells us that if we don't count this offset, we will have a deviation of 25 seconds. If a MapReduce job runs several hours, this 25 seconds could be ignored. In our experiments, we did't consider this offset, because our maximum number of nodes is 16. This small number didn't give rise to evident deviation.

Further more, we mentioned that the increase of number of nodes n will result in slight increase of copy time $t_c(n)$ because of more Http connection setup and teardown. If we don't treat it the same as the handling of downgraded partial copy, it brings about deviation. As for our experiment, the gap between 4 nodes and 16 nodes is too small, so that there is almost no change on $t_c(n)$ when n is increased from 4 to 16. Therefore, this is not a good reason that bring about deviation for our experiment. But for large scale of increasing n, for example changing n from 32 to 1024, the change on $t_c(n)$ is not ignorable.

5.5 Summarize learning workflow

A complete learning process should achieve at least two goals: (1) accumulate learning experience; (2) decide the valid boundary of this learning experience. We need at least three experiments to estimate the execution time of a big problem:

1. An experiment using customized MapReduce program on small scale of data and small scale of computing nodes.
2. An experiment using empty MapReduce program on small scale of data and small scale of computing nodes.
3. An experiment using empty MapReduce program on all available nodes.

The first experiment aims at obtaining the main experience information. The second and the third experiment are used to detect the boundary of learning experience. If any bottleneck occurs, special form of models must be used. Based on these knowledge, we summarize the learning workflow as follows:

1. Fairly sample a part of nodes and data.
2. Apply an empty MapReduce program and customized MapReduce program to sampled nodes. Empty program is used to detect whether copy subphase meets any bottleneck and customized program is used to accumulate main learning experience.
3. Apply an empty MapReduce program to all available nodes. Compare the results of previous empty program experiment and current one. Judge whether we meet bottleneck according to this comparison.
4. Decide which model to use according to the step3's test result and accuracy requirement.

Chapter 6

Conclusion

We started this thesis report by describing the research problem. Then we gradually revealed the mechanism of Hadoop MapReduce. Review our whole report, we mainly did three things:(1) proved execution time of map/reduce tasks follow regular trend under certain conditions;(2) proposed two models to estimate execution time of a MapReduce job;(3) fulfilled scalability analysis.

MapReduce is essentially a parallel programming model, thus we use speedup to measure its performance. We emphasized that the utilization of shared resources is the cause of speedup bottleneck. For a MapReduce job, speedup bottleneck only occurs to copy operation. We name the two copy behaviors which manifest bottleneck as *partial copy* and *downgraded full copy*. An empty MapReduce program is proposed to handle these bottlenecks. If bottleneck is detected, bottleneck form of proposed models has to be used to estimate execution time. In order to address how learning process is performed, we proposed a learning workflow. It serves as a guidance of how to carry out a streamlined learning. Based on the virtual infrastructure service provided by Amazon Web Service, we fulfilled experiment verification for some models. The experiment result further boosts us to fulfil scalability analysis. It reveals the most important conclusion in this thesis: under certain conditions, we can tune MapReduce to achieve $T(nodes, datasize) \propto \dfrac{datasize}{nodes}$. This expression further reveals that, MapReduce can be tuned to achieve linear scaling.

From our research we see that MapReduce is a simple and powerful parallel programming framework. Grasping the discipline of MapReduce can greatly help us carry out large scale of experiment on MapReduce.

Appendix A

Raw experiment results

Our raw experiment results are shown in the following tables. The first row of each table represents the start time of first wave of tasks. The second row represents the end time of first wave of tasks, and it also represents the start time of second wave of tasks. The time unit of the following results is second.

map chain0	**map chain1**	**map chain2**	**map chain3**	**red chain0**	**red chain1**	**red chain2**	**red chain3**
1	1	0	3	345	343	344	346
77	83	83	84	357	356	356	358
156	167	169	169	367	368	368	371
239	250	252	253				
315	340	299					

Table A.1: WordCount,929MB,4nodes

map chain0	**map chain1**	**map chain2**	**map chain3**	**red chain0**	**red chain1**	**red chain2**	**red chain3**
4	0	4	4	654	653	654	653
82	81	82	84	667	665	667	671
165	163	162	165	672	677	679	684
247	245	243	242	685	689	690	695
330	327	326	322	697	701	703	708
411	410	407	402				
489	492	487	483				
568	575	570	564				
650							

Table A.2: WordCount,1.86GB,4nodes

map chain0	**main chain1**	**map chain2**	**map chain3**	**map chain4**	**main chain5**	**map chain6**	**map chain7**
1	1	0	1	1	2	1	1
80	80	81	82	83	84	86	88
158	160	160	165	170	169	133	
red chain0	**red chain1**	**red chain2**	**red chain3**	**red chain4**	**red chain5**	**red chain6**	**red chain7**
172	173	173	174	171	173	174	174
184	185	185	185	186	186	186	186

Table A.3: WordCount,929MB,8nodes

map chain0	**map chain1**	**map chain2**	**map chain3**	**map chain4**	**map chain5**	**map chain6**	**map chain7**
1	1	2	3	0	2	1	2
79	79	81	81	83	83	86	84
157	158	159	161	163	168	168	170
237	237	239	243	246	248	254	257
316	317	319	323	310	332		
red chain0	**red chain1**	**red chain2**	**red chain3**	**red chain4**	**red chain5**	**red chain6**	**red chain7**
333	333	334	333	334	335	335	334
345	345	347	348	349	348	348	349
357	358	359	360	361	363	364	364

Table A.4: WordCount,1.86GB,8nodes

map chain0	**map chain1**	**map chain2**	**map chain3**	**map chain4**	**map chain5**	**map chain6**	**map chain7**
0	1	1	4	1	7	7	1
77	81	78	79	82	84	85	81
157	157	156	161	161	163	161	164
234	235	236	239	241	241	243	243
313	317	315	318	322	321	324	304
map chain8	**map chain9**	**map chain10**	**map chain11**	**map chain12**	**map chain13**	**map chain14**	**map chain15**
1	1	1	4	7	1	5	46
82	84	83	87	84	81	94	123
166	166	164	168	168	169	186	201
249	250	251	251	252	268	279	283
333	334	333	319				
red chain0	**red chain1**	**red chain2**	**red chain3**	**red chain4**	**red chain5**	**red chain6**	**red chain7**
337	337	337	337	338	338	338	338
349	349	349	349	349	350	350	350
361	362	362	362	362	362	362	362
red chain8	**red chain9**	**red chain10**	**red chain11**	**red chain12**	**red chain13**	**red chain14**	**red chain15**
338	339	339	339	341	337	339	340
351	351	351	352	353	352	355	355
363	363	363	364	365	366	369	370

Table A.5: WordCount,3.72GB,16nodes

map chain0	**map chain1**	**map chain2**	**map chain3**	**red chain0**	**red chain1**	**red chain2**	**red chain3**
0	1	0	1	157	158	158	156
16	18	19	18	197	197	198	199
35	36	40	36	237	240	237	242
52	54	61	54	273	279	280	284
73	73	81	73	307	319	322	326
91	91	98	94				
111	112	115	112				
130	131	134	133				
148	152	152	154				

Table A.6: Terasort,2GB,4nodes

map chain0	map chain1	map chain2	map chain3	red chain0	red chain1	red chain2	red chain3
1	3	0	1	329	329	334	331
18	21	20	20	372	372	377	375
37	42	40	41	415	417	420	421
57	64	62	62	454	460	462	466
76	83	83	83	496	500	504	514
93	105	102	102	538	545	547	562
112	126	123	125	581	588	589	609
130	149	144	146	624	632	632	657
148	168	165	166	670	677	679	700
167	187	185	187				
189	210	204	206				
211	230	223	225				
222	248	239	244				
250	268	259	264				
269	291	285	285				
290	309	304	304				
312	321	324	327				

Table A.7: Terasort,4GB,4nodes

map chain0	map chain1	thread2	thread3	thread4	thread5	thread6	thread7
1	3	2	2	3	3	0	3
18	18	19	18	19	20	18	19
35	36	36	35	37	37	36	38
52	54	55	53	54	55	54	55
72	72	72	73	73	73	75	75
red chain0	**red chain1**	**thread2**	**thread3**	**thread4**	**thread5**	**thread6**	**thread7**
78	79	80	78	79	80	80	81
119	122	123	122	122	123	124	123
161	162	162	163	164	165	166	169

Table A.8: Terasort,2GB,8nodes

map chain0	map chain1	map chain2	map chain3	map chain4	map chain5	map chain6	map chain7
0	1	0	2	1	2	2	2
16	17	16	17	17	19	19	19
35	35	35	36	35	36	37	37
53	53	53	54	53	54	55	54
70	72	71	72	71	72	74	74
90	91	89	91	90	92	92	92
108	108	109	109	109	110	110	110
126	126	126	127	126	128	128	129
144	144	144	145	146	146	147	147
red chain0	**red chain1**	**red chain2**	**red chain3**	**red chain4**	**red chain5**	**red chain6**	**red chain7**
149	149	148	150	149	149	150	151
195	192	192	196	191	192	193	197
240	235	236	236	237	237	235	240
276	277	278	279	283	283	281	286
312	320	321	326	328	328	330	335

Table A.9: Terasort,4GB,8nodes

map chain0	**map chain1**	**map chain2**	**map chain3**	**map chain4**	**map chain5**	**map chain6**	**map chain7**
1	1	1	1	1	0	2	1
18	18	18	18	18	17	19	19
36	37	35	36	35	35	37	36
54	54	53	56	53	53	56	57
72	72	74	73	73	72	75	75
89	90	90	90	91	91	93	93
107	108	108	108	109	109	110	111
125	126	126	126	127	128	129	129
144	144	144	145	145	146	147	147
map chain8	**map chain9**	**map chain10**	**map chain11**	**map chain12**	**map chain13**	**map chain14**	**map chain15**
3	0	0	2	2	3	3	1
19	20	19	22	20	20	20	20
39	38	36	42	39	39	40	39
55	57	56	61	58	60	59	58
74	75	76	80	76	79	78	77
93	93	94	97	94	96	95	95
111	112	112	115	113	114	115	114
129	130	131	135	131	133	132	131
148	148	149	149	150	151	152	153

Table A.10: Terasort,8GB,16nodes

Bibliography

[1] Amdahl's law. [Online]. Available: http://en.wikipedia.org/wiki/Amdahl's_law

[2] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[3] Amazon web service. [Online]. Available: http://aws.amazon.com/

[4] Bubble sort. [Online]. Available: http://en.wikipedia.org/wiki/Bubble_sort

[5] Cloud computing. [Online]. Available: http://en.wikipedia.org/wiki/Cloud_computing

[6] J. Dean, S. Ghemawat, and G. Inc, "Mapreduce: simplified data processing on large clusters," in *In OSDI04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*. USENIX Association, 2004.

[7] Elastic computing cloud. [Online]. Available: http://aws.amazon.com/ec2/

[8] A. Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures," *IEEE Parallel Distrib. Technol.*, vol. 1, pp. 12–21, August 1993.

[9] granularity. [Online]. Available: http://en.wikipedia.org/wiki/Granularity

[10] graysort. [Online]. Available: http://sortbenchmark.org/

[11] Gustafson's law. [Online]. Available: http://en.wikipedia.org/wiki/Gustafson's_law

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

[13] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi, "Evaluating mapreduce on virtual machines: The hadoop case," in *Proceedings of the 1st International Conference on Cloud Computing*, ser. CloudCom '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 519–528.

[14] *IEEE 802.3i(10BASE-T)*, IEEE Std.

[15] Amazon instance. [Online]. Available: http://aws.amazon.com/ec2/#instance

[16] K. Kim, K. Jeon, H. Han, S.-g. Kim, H. Jung, and H. Y. Yeom, “Mrbench: A benchmark for mapreduce framework,” in *Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 11–18.

[17] V. Kumar and A. Gupta, “Analyzing scalability of parallel algorithms and architectures,” *J. Parallel Distrib. Comput.*, vol. 22, pp. 379–391, September 1994.

[18] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, 2010.

[19] Mapreduce. [Online]. Available: http://en.wikipedia.org/wiki/MapReduce

[20] mesh network. [Online]. Available: http://en.wikipedia.org/wiki/Mesh_networking

[21] Message passing interface(mpi). [Online]. Available: http://en.wikipedia.org/wiki/Message_Passing_Interface

[22] Newyork. [Online]. Available: http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-super-computing-fun/

[23] O. O’Malley, “Terabyte sort on apache hadoop,” May 2008.

[24] ——, “Winning a 60 second dash with a yellow elephant,” April 2009.

[25] W.-C. Shih, S.-S. Tseng, and C.-T. Yang, “Performance study of parallel programming on cloud computing environments using mapreduce,” in *Information Science and Applications (ICISA), 2010 International Conference on*, april 2010, pp. 1–8.

[26] Hadoop mapreduce configuration file. [Online]. Available: http://hadoop.apache.org/mapreduce/docs/current/mapred-default.html

[27] speculative execution. [Online]. Available: http://developer.yahoo.com/hadoop/tutorial/module4.html

[28] Data synchronization. [Online]. Available: http://en.wikipedia.org/wiki/Data_synchronization

[29] E. Tambouris and P. v. Santen, “A methodology for performance and scalability analysis,” in *Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics*, ser. SOFSEM ’95. London, UK: Springer-Verlag, 1995, pp. 475–480.

[30] T. White, *Hadoop:The definitive guide*, M. Loukides, Ed. O’Reilly Media, 2009.

[31] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42.