Pipelined execution of stages in Apache Spark

Truong Duc Kien

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisor: Prof. Paolo Romano

Examination Committee
Chairperson: Prof. António Manuel Ferreira Rito Silva
Supervisor: Prof. Paolo Romano
Member of the Committee: Prof. João Coelho Garcia

October 2016
To my family
Acknowledgments

In the first place, I would like to express my gratitude to my thesis advisor Professor Paolo Romano at Instituto Superior Tecnico for his continuous support of my research. His patience, enthusiasm and knowledge helped me immensely whenever I got stuck. This thesis could not be completed without his guidance.

In addition, I would like to thank Zhongmiao Li, my labmate for the discussion that we had and for his valuable insights into my project. I also want to thank my fellow EMDC classmates for their encouragement during this thesis.

Last but not the least, I would like to thank my family for their moral support throughout writing this thesis.
Resumo

Esta dissertação visa a investigação da eficiência de uma base fundamental para a construção de plataformas de processamento de big-data modernas, tais como o Apache Spark. Este tipo de plataformas suportam tarefas complexas de análise de dados, permitindo que os cientistas de dados expressem manipulações arbitrárias aos dados, através de um grafo directo de operadores de transformações, executadas de forma distribuída em várias máquinas.

Algumas soluções recentes, como a plataforma Spark acima mencionada, empregam uma estratégia baseada em lotes. Nesta estratégia, os operadores que induzem a troca de dados entre máquinas são utilizados para marcar o início de uma nova etapa, que agrupa logicamente o conjunto de operadores que podem ser executados sem que exista a necessidade de estes requerirem dados de operadores executados em máquinas diferentes. Uma aplicação Spark é baseada em lotes, sendo que se a etapa \( i + 1 \) requer o resultado da etapa \( i \), então a etapa \( i \) deve terminar a sua computação antes que a etapa \( i + 1 \) possa ter início e carregar os dados da etapa \( i \). Esta escolha abordagem pode levar à utilização sub-óptima dos recursos do cluster. Uma abordagem alternativa consiste no encadeamento da transmissão de dados entre etapas, tal que a etapa \( i + 1 \) possa iniciar o processamento de dados à medida que estes são gerados pela etapa \( i \). Esta tese estende o Apache Spark, permitindo a execução de etapas de forma sobreposta, e compara as duas estratégias apresentadas.

Palavras-chave: análise de dados, big-data, Spark, lote, encadeamento
Abstract

This dissertation investigates the efficiency of a fundamental, low-level building block of modern big-data processing platforms, like Apache Spark. This type of platform supports complex data analysis tasks by allowing data scientists to express arbitrary data manipulations via a direct graph of, so called, transformation operators that executed in a distributed fashion across a cluster of machines.

Some state-of-the-art solutions, like the aforementioned Spark platform, take what can be defined a “batch” strategy. Operators that induce exchange of data among machines are used to demarcate the beginning of a new stage, which serves to logically group a set of operators that can execute without requiring any input from operators running on different machines. A Spark application is executed in batch mode, in the sense that if the stage $i+1$ requires the result of stage $i$, then stage $i$ needs to finish its computation before stage $i+1$ can begin and load the data from stage $i$. This choice, however, can lead to sub-optimal utilization of cluster’s resources. Another alternative approach consists of pipelining data transmission between stages, so that stage $i+1$ can start processing the data as it is generated by stage $i$. With this strategy, the execution of the two stages are overlapped, allowing higher utilization of cluster’s resources. In this thesis, Apache Spark is extended to support the pipeline execution. An extensive experimental evaluation was conducted using both synthetic and realistic workloads to evaluate the performance difference between batch and pipelined execution.

Keywords: data-analytic, Spark, big-data, batch, pipeline
# Contents

Acknowledgments ......................................................... v
Resumo ........................................................................ vii
Abstract ......................................................................... ix
List of Tables ................................................................... xiii
List of Figures ................................................................... xv
Glossary ........................................................................... xvii

1 Introduction ................................................................. 1
  1.1 Motivation ............................................................. 1
  1.2 Overviews of big data analytic platforms .................... 2
  1.3 Objectives ............................................................. 4
  1.4 Thesis Outline ......................................................... 4

2 Background .................................................................. 5
  2.1 Related works ......................................................... 5
    2.1.1 Brief history of cluster computing and big-data analytic platforms .... 5
    2.1.2 Modern big-data analytic platforms ............................. 6
  2.2 Apache Spark Description ........................................ 9
    2.2.1 Resilient Distributed Dataset ................................. 9
    2.2.2 Spark programming interfaces ............................... 9

3 Implementation ............................................................ 15
  3.1 Spark’s architecture ............................................... 15
    3.1.1 Master ........................................................... 15
    3.1.2 Worker ......................................................... 15
    3.1.3 Driver ........................................................ 16
    3.1.4 Executor ..................................................... 17
  3.2 Spark’s components ................................................. 17
    3.2.1 The DAGScheduler ......................................... 17
    3.2.2 The TaskScheduler ......................................... 19
    3.2.3 The CoarseGrainedSchedulerBackend ................... 20
    3.2.4 The Map’s output tracker .................................... 21
3.2.5 The block manager .................................................. 21
3.3 Extending Spark for coarse-grained pipelined stage execution ................. 22
  3.3.1 Early scheduling of the consumer stage .................................. 23
  3.3.2 Processing incomplete shuffle data ..................................... 24
  3.3.3 Over-scheduling executors ........................................... 26
  3.3.4 Fault-tolerance implications ........................................ 26

4 Evaluation & Discussion .............................................. 29
  4.1 Environment ......................................................... 29
    4.1.1 Hardware ....................................................... 29
    4.1.2 Software ...................................................... 29
  4.2 Methodology ....................................................... 30
  4.3 Micro-benchmark: Randomize Text ...................................... 30
    4.3.1 Description ..................................................... 30
    4.3.2 Result & Discussion ........................................... 31
  4.4 Classic two-stage cluster processing benchmark ............................... 36
    4.4.1 Words Count .................................................. 36
    4.4.2 Terasort ....................................................... 37
  4.5 Multi-stages benchmarks .............................................. 40
    4.5.1 PageRank ..................................................... 40
    4.5.2 Connected Components ......................................... 42
    4.5.3 Triangles Count ............................................... 44

5 Conclusions .......................................................... 51
  5.1 Future work ......................................................... 52
    5.1.1 Fine-grained pipeline shuffle .................................... 52
    5.1.2 Memory-aware scheduler ....................................... 52
    5.1.3 Support other types of cluster manager .......................... 52

Bibliography .......................................................... 53
## List of Tables

4.1 Randomize Text Execution Statistics .............................................. 34
4.2 Words Count’s Execution Statistics ............................................. 37
4.3 Terasort’s execution statistics .................................................. 39
4.4 PageRank Execution Statistics .................................................. 42
4.5 Connected Components Execution Statistics ............................... 45
4.6 Triangles Count Execution Statistics ....................................... 48
## List of Figures

1.1 Two-Stage Batch Data Transfer ............................................... 3
1.2 Fine-grained Pipeline Data Transfer ........................................ 4

2.1 Spark driver and workers ....................................................... 10
2.2 Spark’s words count RDD & transformations .............................. 12
2.3 Spark’s dependency types ...................................................... 13

3.1 Spark’s resource allocation .................................................... 16
3.2 Spark’s DAG construction ..................................................... 18
3.3 Spark’s tasks ................................................................. 19
3.4 Map output tracking process .................................................. 22
3.5 Tasks’ execution deadlock ..................................................... 23
3.6 Correct barrier-less stage scheduling ...................................... 24
3.7 Map output trackers’ communication ....................................... 25
3.8 Shuffle fetcher iterator ....................................................... 26
3.9 Partial shuffle fetcher iterator ............................................... 27

4.1 Timeline of Active Executors for Randomize Text ........................ 31
4.2 Comparison of Randomize Text Execution Time - 4 Iterations ......... 32
4.3 Comparison of Randomize Text Execution Time - 8 Iterations ........ 32
4.4 Comparison of Randomize Text Execution Time - 16 Iterations ....... 32
4.5 Comparison of Randomize Text Execution Time - 0.6 GB ............... 33
4.6 Comparison of Randomize Text Execution Time - 1.2 GB ............... 33
4.7 Comparison of Randomize Text Execution Time - 1.8 GB ............... 34
4.8 Randomize Text Execution Timeline for 0.6 GB Input .................. 35
4.9 Timeline of Active Executors on modified Spark for Words Count ...... 36
4.10 Comparison of Words Count Execution Time ............................. 37
4.11 Words Count’s Execution Timeline for the 9.6 GB document .......... 38
4.12 Timeline of Active Executors on modified Spark for Terasort ........ 38
4.13 Comparison of Terasort Execution Time .................................. 39
4.14 Terasort’s Execution Timeline for the 6.4 GB input ..................... 40
4.15 Timeline of Active Executors on modified Spark for PageRank ....... 41
4.16 Comparison of PageRank Execution Time ........................................... 42
4.17 PageRank’s Execution Timeline for the 3 millions vertices graph ............... 43
4.18 Timeline of Active Executors on modified Spark for Connected Components .... 44
4.19 Comparison of Connected Component Execution Time .......................... 45
4.20 Connected Components Execution Timeline for the 3 millions vertices graph .... 46
4.21 Timeline of Active Executors on modified Spark for Triangles Count .......... 47
4.22 Comparison of Triangles Count Execution Time .................................... 47
4.23 Triangles Count Execution Timeline .................................................. 49
Glossary

**DAG** Directed Acyclic Graph is a directed graph with no cycle.

**RDD** Resilient Distributed Dataset is the data structure to represent data in Apache Spark.
Chapter 1

Introduction

1.1 Motivation

As technology improves, more and more data are being produced and stored every day. Along with the increase in volume of data is the rise in computational power needed to analyze these data. It is no longer adequate to process information using a single computer. As a result, the majority of the newly-invented data processing systems make use of cluster computing. These systems orchestrate multiple computers at the same time in order to work on the massive amount of input data that no single-computer system can handle. However, cluster computing poses its own challenges.

In the first place, the application has to be re-written to take advantage of the high parallelism that a cluster provides. As distributed programming is very complicated to use, cluster-computing systems have to provide a highly flexible programming model that can easily be used to efficiently implement many algorithms. Thanks to their adaptability, Google Map/Reduce [1] and Apache Spark [2] have risen to become some of the most popular data processing systems in recent years.

In addition, the issue of fault-tolerance is also very important with cluster computing. These faults include both catastrophic failures and stragglers. Catastrophic failures happen when a machine stops functioning completely, while stragglers are when machines work slower than normal due to outside interference. Among these two, stragglers are particularly cumbersome because they occur more frequently and are harder to detect. This is further exacerbated in a cloud computing environment. In this environment, computers are often virtual machines (VMs), which naturally receive a higher degree of interference due to the co-location of multiple VMs on the same physical host. The heterogeneity of cloud computing, combined with the stragglers, means a lot of computing resource might be wasted if the entire cluster has to wait for a few slower machines. Unfortunately, in Map/Reduce and Spark, the machines in cluster are often required to block and synchronized, when data need to be redistributed between computers. This problem is especially prominent in Spark, since a Spark program can contain many more barriers than a Map/Reduce program.
1.2 Overviews of big data analytic platforms

A cluster computing framework composes of two layers: the application programming interface (API) and the underlying execution engine - also called the runtime. In order to use the framework, a developer has to write applications that contain customized logic to manipulate data using the provided API. Some frameworks also provide predefined functions for common tasks such as sorting, grouping, or filtering data to ease development effort. We will refer to both the user-defined functions and the provided utilities as operators in this document. A data analytic job is implemented by combining these operators in a coherent order. This order specifies the dependencies between operators: an operator can consume the output of different operators, and in turn generates input for other dependent operators.

How operators’ dependencies are defined varies from frameworks to frameworks. In Map/Reduce [1][3], the user can only create two operators: a Map operator and a Reduce operator. Inside a Map/Reduce job, the operators’ dependencies are implicit: the Reduce operator always runs after and consumes the output of the Map operator. On the other hand, in more recent frameworks, like Flink [4] or Spark [2], there are no restriction on how many operators can be used in a program. As a result, the relationships between operators have to be explicitly defined by developers.

Once a program is created using the data analytic framework’s API, it is executed on the cluster by the framework’s execution engine. As the data is processed, it needs to be transferred from an operator to another. Apart from a few embarrassingly parallel workloads, most programs require data to be exchanged not only between operators in the same machine, but also between operators running on different computers in the cluster.

In many state-of-the-art platforms, such as Map/Reduce and Spark, when an inter-computer data transfer is required, it is performed by a two-stage batch shuffle operation, where the output of one operator are partitioned and sent to other computers in the cluster based on the key of each element in the output. As shown in figure 1.1, during the first stage of a two-stage batch shuffle, all sending operators are required to store shuffle data to a buffer, and a synchronization barrier is used here to ensure the completion of all senders. In the second stage, the data is fetched and processed by the receiving operators. Due the existence of the barrier between stages, the faster machines in the cluster have to wait idly for the slower machines to catch up.

An alternative to the batch shuffle, shown in figure 1.2, is the pipeline shuffle, where the records are transferred from the sender operator to the receiver operator as they are processed, without having to wait for the sender operator to finish. This design does not requires the use of a stage barrier, and are often used in real-time streaming systems like S4 [5], Storm [6], and Flink [4], where the processing time of each record have to meet a strict deadline. Still, in other use-cases, the batch shuffle used in Map/Reduce and Spark offers many advantages over a pipeline system.

In the first place, because the output of a stage is grouped into large chunks before transmitting to other computers in the cluster, the number of network connections and their overhead are minimized. Due to the large block size, the transferred data can also be compressed with a higher ratio than in a pipeline system, reducing the amount of data needed to send over the wire. In a computer cluster,
networking is often the slowest part of the system, so being able to use network efficiently is imperative to achieve a high data processing throughput.

In addition, being able to capture a snapshot of the system makes implementing fault-tolerance much easier. In a batch shuffle, a checkpoint can be performed by persisting the output buffer to disk, as done in Spark. Any failed subsequent stages can be restarted by reading from this on-disk shuffle file. In a pipeline shuffle, it's much more difficult to do a snapshot, since the state of the network channel between computers also needs capturing, which is a non-trivial task.

Lastly, a batch shuffle system completely decouples the sending and receiving of data, thus does not require the producer and the consumer stages to be run at the same time. This simplifies the scheduler, because only the operators in the current stages have to be factored in the calculation. In a pipeline shuffle, the scheduler have to make sure that the operators in both the producer and the consumer stages are active at the correct order to avoid deadlocks. In applications that have many shuffles, where a stage can be both a producer and a consumer of shuffle of data, a pipeline system will require the scheduler to create a scheduling plan for all operators in the application at once.
1.3 Objectives

Despite all the advantages of a batch shuffle system, the presence of a stage barrier wastes CPU times in faster computers and reduces system’s efficiency. The goal of this thesis is to investigate the impact of using a pipeline execution strategy in Apache Spark by removing the stage barrier, both in terms of implications on correctness of the computation and on fault-tolerance. Furthermore, via an extensive experimental evaluation using both synthetic and realistic workloads, we measured the performance difference of Apache Spark when switching from batch execution to pipeline execution.

Although, the problematic stage barrier exists in both Map/Reduce and Spark shuffle system, the later was chosen as the research target because a Spark program is more complex and can contain many barriers, while a Map/Reduce program only has one. This difference means Spark is both more susceptible to slow down and can be more noticeably improved if the CPU idle time caused by the barrier is used more effectively.

1.4 Thesis Outline

The rest of the thesis is structured as followed:

- Chapter 2 provides the background information needed to understand this thesis. It includes a summary on the current state-of-the-art cluster processing frameworks and an overview of Apache Spark.

- Chapter 3 explains the architecture of Apache Spark’s and how it has to be modified to implement a pipeline execution strategy.

- Chapter 4 describes our evaluation to determine the impact on performance of pipeline data transfers in Spark. We performed benchmarks with many algorithms with different workload to measure the difference in execution time of a Spark application when running in our modified runtime and the original Spark.

- Chapter 5 provides the conclusion of this thesis and gives some insights into future works that can be used to improve Spark further.
Chapter 2

Background

2.1 Related works

2.1.1 Brief history of cluster computing and big-data analytic platforms

Cluster computing is not a recent invention: initial efforts to use multiple computers working in coordination to solve problems, which cannot be solved by one computer, appeared in the 1960s [7]. However, these initial clusters were often ad-hoc: designed and built by the users for one specific application. It was not until the 1980s that a general purpose computer clusters become commercially successful with the appearance of DEC’s VAXcluster system [8]. Surprisingly, in 2016, 40 years later, an evolved version of VAXcluster are still being developed and used with the name VMSc cluster. The first VAXcluster consisted of computers running the VMS operating system, connecting through specialized hardware and software. These computers are “closely-couple”: separate processors and memories with message passing as the mean of interconnect, but all computers in the cluster need to be physically closed together to achieve high speed connections. These characteristics are still common to many modern computer clusters.

However, the use of vendor specific hardware and software soon proved to be too restrictive. As many people started working on developing platforms and abstractions aimed at simplifying harnessing the processing power made available by large scale, and potentially highly heterogeneous clusters, MPI - Message Passing Interface [9] and PVM - Parallel Virtual Machine [10] are probably some of the earlier solutions in this sense, having appeared in the early 1990s. The PVM software is developed by Oak Ridge National Laboratory and has to be installed in all nodes in the cluster. It provides a run-time environment to facilitate message-passing, task and resource management, and fault notification. On the other hand, MPI was started as a collaboration between many organizations, including universities and computer manufacturers. Therefore, it combines the most useful features of several systems at the time. In addition, instead of being a concrete software implementation, MPI is a standard which defines the semantics of a core of library routines that are useful to a wide range of distributed application. This has two major benefits. Firstly, vendors can develop their own version of software that supports the MPI standard but is optimized for their systems. Secondly, the users can write a portable program using
the MPI’s syntax and run that on different clusters without modification. Thanks to these advantages, although MPI appeared later, it has long surpassed PVM in popularity.

Despite being the de-facto standard in cluster computing for a long time, the MPI standard was quite low-level and difficult to use. Furthermore, MPI has no mechanism to provide fault-tolerance: if a node goes down, the entire application crashes. This makes MPI only suitable in controlled environments where computers and network are extremely reliable. These highly available cluster are expensive and difficult to build and maintain, especially when clusters are becoming larger and larger to handle the increasing amount of data. Over the last decade, we have seen a proliferation of cluster computing platforms, like Map/Reduce [1] and Dryad [11], aimed at simplifying the development of large-scale data analytic applications to run on cheap, unreliable, commodity hardware. The rise of cloud computing, in which computer clusters are created with virtual machines - an even more unreliable environment, further increases the necessity of these new data processing frameworks. Another key factor, that has helped the modern cluster computing systems to gain their popularity, is that they often provide a high-level programming interface that simplify applications’ creation. While this behavior often reduces the maximum achievable performance, it is offset by the ability to use larger clusters of cheaper hardware.

2.1.2 Modern big-data analytic platforms

Hadoop Map/Reduce [3] might be the most well-known modern data processing framework. It is an open-source implementation of the Map/Reduce programming model popularized by Google [1]. The Map/Reduce model is based on the divide-and-conquer method: a big problem is divided into multiple sub-problems that are small enough to be solved directly in a single computer. These sub-problems are then processed independently in parallel in different nodes of the cluster. Finally, the solution to these sub-problems are merged to get the result of the big problem.

The execution of a Map/Reduce program is divided into two distinct stages. During the first stage, the data is distributed to all machine in the cluster to process and generate results in the form of key-value pairs. In the second stage, results are grouped by key and all pairs with the same key are collected and analyzed at the same computer. The users only have to supply one function to run in each phase to work on the input, while everything else is handled by the framework. This is much simple than writing a similar program using MPI or other low-level tools, although at the cost of having to adopt a less expressive programming paradigm.

Although the simplicity and genericity of Map/Reduce’s programming interface allows developers to implement many algorithms, not all distributed computations/data analytic jobs fit nicely the programming paradigm of Map/Reduce, which forces all applications to be expressed as the sequential application of a series of map and reduce phases. In order to implement such algorithms, multiple Map/Reduce programs have to be chained together, which incurs a lot of overhead due to the cost of persisting the output of each Map/Reduce program to disk. As a result, many newer frameworks have been created to overcome these drawbacks, such as Dryad [11], Tez [12], Spark [2] and so on.

Dryad [11] is another cluster computing framework developed by Microsoft. A Dryad application
is modeled as a directed acyclic graph (DAG) in which the computational tasks are performed in the
vertices and the communication channels are represented as edges. In order to use Dryad, the pro-
grammer have to write the programs to run at the vertices and connect them together using channels.
These programs read data from their input channels, perform calculation then write the result to the
output channels. Dryad’s runtime will take care of generating the DAG and executing it on the cluster
by assigning the programs to the computers and mapping the logical channel to a suitable back-end:
shared files, in-memory pipes or TCP/IP connection etc. Compared to Map/Reduce, Dryad is highly flex-
ible. Firstly, a Dryad application can specify a variety of communication patterns, while Map/Reduce is
restricted to the scatter - gather topology. Secondly, a Dryad application can have many operators, each
can have multiple input and output channels. On the other hand, a Map/Reduce only have two operators
with strict input and output requirements. This flexibility mean many algorithms are easier to implement
in Dryad than in Map/Reduce. Unfortunately, Microsoft stopped developing Dryad and decided to focus
on Hadoop Map/Reduce in 2011.

**Tez** [12] is an open-source framework to build complex dataflow-centric runtime Instead of only pro-
viding tools to write users’ applications, the main purpose of Tez is supplying scaffolding and libraries to
write the execution engine itself. Tez allows users to model computation as a DAG - similar to Dryad, but
with finer control and customization of the edges and vertices. In addition, it exposes API to access and
modify the DAG to support dynamic optimization. Building on top of Hadoop Yarn [13], a key compo-
nents of Hadoop Map/Reduce, Tez also provides many state-of-the-art features such as security, data
locality awareness, resource reuse and fault tolerance.

**S4** [5] is a general-purpose, distributed scalable, and fault-tolerant stream processing platform de-
veloped by Yahoo!. Its main objective is to provide an easy way to develop application that can process
continuous, unbounded stream of data. The data in S4 system is represented using events, in the form
of key-value pairs - similar to Map/Reduce. The operators in a S4 application are called the **Processing
Elements** (PE), which are implemented by the programmer. The system will automatically initialize one
processing element for each key to process all events with that key. On receiving and processing an
event, the PE can either emit more events for further processing by other PEs, or publish the result to
external systems. The main advantages of S4 over Map/Reduce are lower latency and the ability to
handle unbounded data, thanks to its pipeline design.

**Storm** [6] is also a distributed stream processing engine, similar to S4. However, there are several
major differences. Firstly, while a S4 data stream consists of key-value pairs, a Storm data stream
is made of tuples, which is a more flexible programming model. Secondly, the operators in Storm,
called bolts, have to handle unrelated tuples that exist in a stream of events: there are no group-by-key
performed by the framework like in S4. In addition, the number of operators have to be configured by the
users and not automatically determined by the framework. Thirdly, in Storm, an operator pulls the event
from the source, while S4’s operators push the events to the receiver. This difference means a Storm
application is less vulnerable to buffer-overflow errors like in S4. Finally, Storm provides guarantee about
events processing: either "at-least-once" or "at-most-once" depending on the user’s requirement, while
there is no such guarantee in the case of S4. In S4, checkpoints are performed in an uncoordinated
fashion by each node. As a result, when a node fails, events that arrive between the last checkpoint and the recovery time are lost.

**Flink** [4] is an attempt to provide a framework for parallel data processing that support both batch processing like Hadoop and stream processing like Storm, using a common execution runtime. This capability is provided by the runtime's pipeline architecture where data is continuously transformed and flown between operators. In addition, Flink has a very flexible API: it provides multiple methods to perform coarse-grain data transformations, which allows programmer to manipulate and process input data in various ways. Most of these methods accept a user-defined function that can be supplied by the programmer to implement processing logic. Furthermore, unlike Dryad or Tez, a Flink application is not restricted to a DAG topology: Flink's runtime can natively handle an iterative dataflow between operators.

**Spark** [2] started out as a research project in UC Berkeley but has grown to become one of the most popular cluster computing framework in recent years. While Tez and Flink are newer, Spark is much more mature and heavily used in both academia and the industry. This is thanks to its high performance and user-friendliness. Spark has one of the easiest to use API, where the developers can use customized functions to manipulate data and elegantly chain the output of these functions to specify data dependencies. This simplicity speeds up Spark adoption significantly. In fact, Flink decided to switch from their original programming interface to one that looks remarkably similar to Spark's since version 0.6. In term of performance, Spark is a major improvement compared to Map/Reduce. Instead of chaining tens of Map/Reduce programs to implement an algorithm, and paying the expensive price of disk I/O, the developer can implement the entire logic of the algorithm in a single Spark program. In addition, Spark runtime tries to keep the data in resident memory for most transformations, thus significantly reducing the amount of disk accesses. Consequently, algorithms implemented in Spark tend to be much faster than their Map-Reduce counterparts.

Originally designed as a batch processing engine to replace Map/Reduce, Spark has expanded to include stream processing capability. Because the runtime of Spark uses a batch design for shuffling data, an application is divided into multiple stages. These stages are arranged in in a directed acyclic graph and executed topologically. This separation of stages means that the program will never progress pass the first stage without a limit on the input data, making Spark unsuitable for streaming workload. In order to solve this problem, Zaharia et al. proposed a system called **Discretized Stream (DStream)** [14], in which the unbounded input stream is split into smaller batch that can be processed sequentially using the original Spark runtime. This approach is also called micro-batching, because the batches have to be small enough so that the system can run them in seconds to satisfy the stringent latency requirement of stream processing. Once the result of each micro-batch is calculated, the programmer can easily merge them together to calculate the final result if necessary.
2.2 Apache Spark Description

As already mentioned, the target platform used in this dissertation is Apache Spark. In the following I provide an overview about the high-level design of Spark, and its basic programming interface.

2.2.1 Resilient Distributed Dataset

The core of Spark is based on the concept of Resilient Distributed Dataset (RDD), a distributed memory abstraction that allow programmers to perform fault-tolerant data processing across an entire cluster of computers. The characteristic of a RDD is presented in its name:

- **Resilient**: A RDD is designed to tolerate failure, and can easily survive most faults encountered during the application's execution.

- **Distributed**: The data of a RDD is distributed across all machines that participate in the cluster. This partitioning of data ensures the entire cluster's resource is used to process the data and minimize the negative effect of machine failure.

- **Dataset**: A RDD acts as a collection of records. Each record is independent of another and there are no constraints on what can be used as a record except that it must be serializable. This requirement is needed, because a record has to be transformed into a binary form before transmitting over the network to a different machine.

An RDD is an immutable set of records and can only be created through deterministic operations from either the input data or other RDDs. As a result, as long as the input data is stable, the content of a RDD is specified only by these transformations. Thanks to this design, Spark is able to provide fault-tolerance purely by remembering the series of transformation that is used to generate each RDD. This series is also called the **lineage** of an RDD, and is used to re-construct the RDD in case of failure.

In addition, the lineage allows Spark to do lazy-computation: a RDD is only materialized when special operators, which can expose the content of a RDD to external systems, are invoked. These operators are called *actions*. As the cluster often have to keep track of multiple RDDs at once, by not computing an RDD if it is not needed, Spark can keep as much content of other relevant RDDs in main memory for as long as possible. This intelligent caching of data greatly improves the performance of the system.

Finally, users can also ask Spark to cache or checkpoint specific RDDs. Caching enables the developers to selectively keep some RDDs in memory. This is a very powerful capability, because the user can make better judgment than Spark about which RDDs to cache thanks to their knowledge of the algorithm. Checkpointing persists the content of an RDD to stable storage, allowing faster recovery in case of fault.

2.2.2 Spark programming interfaces

Spark provides a library that the developers can use to interact with RDDs in Scala, Java and Python. Generally, the Java library can also be used by other programming languages that run on the Java Virtual
Machine. In order to use Spark, the users have to write a driver program that will be connected to the cluster of workers machine when running as specified in figure 2.1. This driver contains instruction on how each RDDs are created from input data and what the operations needed to turn the input into the output.

Transformations and actions

There are two types of operators in Spark: transformations and actions. Transformations are used to define new RDD from an existing RDD, while actions trigger processing of the RDD to return results to the driver program or making the RDD content available to external systems.

A Spark’s transformation acts as a function that takes in one more more RDDs and produces another RDD. Because RDDs are immutable, a transformation can never modify the input RDDs, the output RDDs are always new, distinct RDDs.

Spark’s transformations are always lazily-evaluated: each RDD is never materialized immediately where the transformation is specified in the driver code. Instead, the occurrence of a transformation causes the lineage of the RDD to be updated with the new transformation. Only when an action is called on an RDD, will Spark trace that RDD’s lineage and perform all transformations needed to compute it.

Listing 1 is an example of a Spark program. Its job is to process an input file and count the occurrence of each unique word and save the result to an output file.

In order to calculate the final result, this program uses 4 transformations from line 3 to line 6, each of which creates a new RDD as illustrated in figure 2.2

The first transformation is sc.textFile(inputFile), which converts the input data into a RDD of strings, with each record represents a line in the input file.

The second transformation is inputRDD.flatMap(...), which takes each line in the inputRDD and split it by the white space character. The output of this transformation is a new RDD of words.

Line 5 contains the third transformation wordsRDD.map(...), which marks each word with the number
val sc = new SparkContext(conf)

val inputRDD = sc.textFile(inputFile)
val wordsRDD = inputRDD.flatMap(line => line.split("\s+"))
val wordsTupleRDD = wordsRDD.map(word => (word, 1))
val wordsCountRDD = wordsTupleRDD.reduceByKey((a, b) => a + b)

wordsCountRDD.saveAsTextFile(outputFile)

Listing 1: Spark’s words count program

1 to generate an RDD of key-value pairs. Each record in this RDD has a key which is the content of the word, and a value which is the number of appearance of the word before aggregation.

The next line has the final transformation wordsTupleRDD.reduceByKey(...), which groups the records in the wordsTupleRDD by the content of the key and sums up the value. This result in the final RDD, which contains the words and their number of appearance in the input documents.

The last line is an action wordsCountRDD.saveAsTextFile(...), which triggers computation of the resulting RDD and saves its content to external files.
Narrow dependencies and wide dependencies

As mentioned in the previous section, transformations are used to create new RDDs from old RDDs, which introduces dependencies between the input and output RDDs. In Spark, these dependencies are divided into two groups: narrow dependencies and wide dependencies, which are illustrated in figure 2.3.

Narrow dependencies happen when each partition of the parent RDD is only used to compute at most one partition of the child RDD. This type of dependency generally occurs in the following cases:

- Using transformations that uniformly change every records of the parent RDD, such as filter, map, flatMap, etc by applying the same user-defined function to each record.
- Merging of RDDs using the union transformation. In this transformation, each partition of the input RDDs will be converted to one partition of output RDD.
- Joining of RDDs when all input RDDs are already partitioned using the same partitioner. In this operation, each partition of the output RDD is the result of one partition from each input RDDs.
These partitions are guaranteed to be on the same computer due to the sharing of partitioner, so no data shuffling is needed to compute the output partition.

Narrow dependencies are very efficient because Spark will co-locate the parent and child partition in the same computer and use shared memory to pipeline the data directly without going through the shuffle.

Contrary to narrow dependencies, wide dependencies means each partition of the parents RDDs is needed by more than one partitions of the child RDD. For example, when you do a reduceByKey transformation, as the key can exist in any partitions of the parent RDD, all of these partitions are needed to generate a partition of the child RDD. Similarly, when joining two RDDs which were partitioned using different partitioners, they have to be repartitioned with the same partitioner so that the items with the same key in both RDD are gathered in the same machine. Consequently, a wide dependency always requires data shuffling and is also called a shuffle dependency.

**Figure 2.3:** Spark’s dependency types: blue rectangles represent the partitions of a RDD, while the arrows indicate dependencies.
Chapter 3

Implementation

In the design of Spark, a shuffle dependency is always executed in at least two stages: one or more stages produce the shuffle data - the producers, and one stage that consumes the shuffle data - the consumer. Currently, the consumer stage is not started until the producer stages have completed. As discussed in the previous chapters, this behavior can lead to idle CPUs while waiting for the last few tasks of the producer stages to finish, wasting the cluster’s resource. In this chapter, I present a method to remove this execution barrier, and thus partially overlap the two side of the barrier in a shuffle dependency.

3.1 Spark’s architecture

Spark uses a master/workers architecture: each Spark cluster has one master node and multiple worker nodes.

3.1.1 Master

The master node is connected to all worker nodes to manage the cluster’s resource and provide the single gateway into the clusters. When running, the driver program, mentioned in 2.2.2, has to contact the master to request for CPUs and memory. The master will in turn allocate resources on the worker in the form of executor processes and connect these executors to the requested driver. This is shown in figure 3.1

3.1.2 Worker

Worker nodes are the places where the actual computation takes places. After starting up, each worker node attempts to connect to the master and register the number of CPU cores and amount of RAM that is available on the machine. Thereafter, every 15 seconds, the worker sends a heart-beat message to notify the master that it is still alive.
On receiving a resource request from the master, the worker will spawn new executor processes according to the request. Executors are not reused or shared between application: each program will have its own executor processes. Once launched, the executors are monitored by the worker and will be killed when commanded by the master. This usually happens when the applications complete, or there is some unrecoverable faults in the cluster.

Apart from executor processes, worker nodes can also house driver process if needed. This is the case when the developers want to use “cluster” deployment mode. In this mode, the master will choose one random worker in the cluster to run the driver process.

### 3.1.3 Driver

The driver is the brain of a Spark’s application. It can be run from outside the cluster using the *client* deployment mode or from within the cluster using the *cluster* employment mode. It manages how the data are distributed and processed on different executors. As the driver needs to constantly communicate with executors on workers, it should ideally be deployed on a machine with fast network connection to the cluster.

In order to perform its jobs, the driver process contains multiple components:

- **The RPC environment**: This is used to communicate with executors for command and control purposes.

- **The scheduler**: The scheduler decides what, where and when the computation is done. The scheduler component consists of the DAGScheduler, the TaskScheduler, the TaskSetManager
and the Scheduler backend.

- **The Map’s output tracker**: The driver’s Map’s output tracker maintains a global index about the locations of the output generated by each map stages.

- **The block manager**: The driver’s block manager contains information about which and where data blocks are stored in the entire cluster.

### 3.1.4 Executor

The executor is the muscle of a Spark application: it receives tasks from driver, runs those tasks and reports results and other statistics back to the driver. Executors are spawned in worker nodes on request from the master and generally live for the entire application unless they crash or are killed prematurely by the master as shown in Figure 3.1. If an executor crashes, another one will be immediately created to replace it. Once an executor starts, it contacts the driver process directly to register the number of free CPU cores it has and waits for task from the executor. Throughout its life span, an executor will be asked to run multiple tasks by the driver.

The main components of an executor are:

- **The RPC environment**: This is used to communicate with the drivers to register resources, accept tasks from driver and updates their statuses back to the driver.

- **The task runner**: This is used to run a task inside a thread pool and collect its statistic.

- **The Map’s output tracker**: An executor’s Map’s output tracker store a cached copy of the driver’s Map’s output tracker for completed stages.

- **The block manager**: An executor’s block manager has information about the location of the data blocks stored in that worker.

### 3.2 Spark’s components

This section will discuss about some main components of Spark’s driver and executors that we have gone over in the previous section.

#### 3.2.1 The DAGScheduler

From the driver program written by the programmer, Spark creates an execution plan consists of multiple jobs, one job for every action performed. A job is created by grouping all transformations performed from the beginning of the program up to that action. The job is then submitted to the DAGScheduler for execution.

The DAGScheduler is the high-level scheduler of Spark that implements stage-oriented scheduling. It computes a DAG (directed acyclic graph) of stages for each job and figures out the minimum schedule
to run the job. This minimum schedule takes into account of what RDDs and stages’ output are available from previous jobs to avoid unnecessary re-computation.

In order to create a DAG of stages, Spark breaks the RDD dependencies graph at shuffle boundaries. As illustrated in figure 3.2, all contiguous operators with narrow dependencies, like filter or map, are grouped together into a single set of tasks and data is pipelined between these operators. On the other hand, operators with wide dependencies like reduceByKey or repartition are broken into two parts: one to write the map output for shuffling and one to read this shuffled data. These two parts will mark the end of a stage and the beginning of the next one. Consequently, each stage will only have wide dependencies on other stages at its boundary, and all operators inside a stage can be pipelined.

Figure 3.2: Spark’s DAG construction. A blue rectangle stands for a partition of a RDD, which is represented by the surrounding rectangle. The dash-lines are used to separate the stages.

Once the DAG is created, Spark will start executing the stage in topological order. For example, with the program in figure 3.2, stage 1 and stage 2 are always executed before stage 3. However, stage 1 and stage 2 can be executed at the same time or serially depending on the availability of the cluster’s resources, because there is no dependency between them. The execution of stages is done by generating one TaskSet for each about-to-be-run stage, and submitting the TaskSet to the underlying TaskScheduler. Each TaskSet consists of multiple independent tasks that can be run in any executors at any worker nodes. The preferred location to launch each task is calculated based upon the amount of input data at each executor: the executors that have the more input data for the task will get higher priority. This strategy is chosen to minimize the amount of data transfer through the network. Because the size of a task are generally much smaller than data, it’s often much faster to send the task to the
executor than to fetch the data to a remote executor.

The final job of the DAGScheduler is to handle failures due to the loss of map output files, in which case the old stages need to be run again to regenerate those files. Because the DAGScheduler knows what map output files are still available, the re-run stages do not have to be re-executed completely: only tasks that created the missing output are necessary.

### 3.2.2 The TaskScheduler

The task scheduler is the low-level scheduler of a Spark application, which receives tasks from the DAGScheduler and allocate them to executors. Additionally, the task scheduler also monitors executors to get the progress of the currently running tasks. This is done by listening to the regular heartbeat signal from every executors, which is sent every 10 seconds.

A task is the smallest execution unit in Spark. Each task is fully independent, self-contained, and identified by the stage id, the stage attempt id and the partition id. In other words, a task represents the computation of a single data partition in an attempt to execute a stage in a Spark job as illustrated in figure 3.3. Depending on the type of dependency between the input RDD and the output RDD, a task can consume one or more partitions of the input RDDs to generate one partition of the output RDD.

![Task Diagram](image)

(a) Narrow dependency

(b) Wide dependency

Figure 3.3: Spark's tasks: each blue rectangle represents an partition of a RDD, while the lines indicate data dependencies.

All tasks with the same stage id and stage attempt id belongs to the same TaskSet, which is created
by DAGScheduler and sends to the task scheduler as mentioned in the previous section. Inside the

task scheduler, the TaskSets are managed independently using separate TaskSetManagers. If multiple

TaskSets are submitted at the same time, the task scheduler will try to run the TaskSetManagers in a

sequential order defined by the scheduling policy. The default ordering is based on the stage id and the

TaskSet’s priority.

A TaskSetManager keeps track of each task in the TaskSet and matches the tasks with the preferred

executors using a technique called delay scheduling [15]. There are 4 locality levels in the order of
descending preference:

- **PROCESS_LOCAL**: The task will be launched in the same executor process as its input data.

- **NODE_LOCAL**: The task will be launched in the same node, but different executor process from

  its input data. This is also used when the task read from external systems like HDFS.

- **RACK_LOCAL**: The task will be launched in different node, but the same rack. This is only used if

  the rack information is configured.

- **ANY**: The task will be launched in any free executors.

All tasks start with the highest locality preference (PROCESS_LOCAL or NODE_LOCAL) and will

wait for a small period to see if any executors match their preference. After this period, the preferred

locality is reduced, so that more executors can be matched. This matching of tasks to executors is done

using a best-first approach: a task will be launched if there are free executors and no other task with a

higher locality preference can be matched to those executors.

In addition to pairing tasks and executors, a TaskSetManager also watches for tasks’ failures and

retry them up to 3 times if possible. In case of unrecoverable errors, the TaskSetManager cleans up

itself and exits gracefully to let the DAGScheduler restart the entire stage.

3.2.3 The CoarseGrainedSchedulerBackend

In order to support multiple type of cluster managers such as Apache Mesos [16], Apache YARN [17] or

Spark’s own Standalone cluster manager, Spark has a pluggable backend mechanism that allows the

mapping of a cluster’s resources to Spark’s executors in a flexible way. In this document, we will only

focus on the **CoarseGrainedSchedulerBackend**, which is used in Spark’s Standalone cluster, because

it is the most popular and well-integrated into Spark.

The CoarseGrainedSchedulerBackend acts as the executors’ repository for a Spark application. It

tracks executors’ registration, de-registration, and maintains a record of free executors and CPUs across

the cluster. This scheduler backend also reuses the same executors for the entire duration of a Spark

application, instead of launching a new executor for each task.
3.2.4 The Map’s output tracker

The MapOutputTracker is the component that tracks the location of shuffle output produced by a stage. For each stage, the map output is divided into buckets, each bucket is generated by a task and corresponds to a partition in the input RDD. There are two versions of MapOutputTracker:

- **MapOutputTrackerMaster**: Runs on the driver and contains the locations of every map output buckets, for all shuffle in the cluster

- **MapOutputTrackerWorker**: Runs on the executors and is a local cached of the map output location to avoid contacting the driver every time a new task is started on the executors. A cache-miss will trigger a re-fetch of the map’s output location from the MapOutputTrackerMaster.

As indicated by figure 3.4, whenever a task finish processing, it has to contact the driver, which will register the location of that task’s map output in the MapOutputTracker. A map output location is defined by:

- The id of the shuffle: every wide-dependency between two stages is assigned a unique identifier, which also acts as the shuffle’s id.

- The id of the data partition that the task compute

- The id of the block manager running on the executor that processed the task

Afterward, when the tasks of the child stage are launched, they will contact the MapOutputTracker to get the location of the parent stages’ map output so that they can fetch and process this data.

3.2.5 The block manager

The block manager is the key-value store that runs in the driver and executors of a Spark application. It provides a unified interface to store and fetch data blocks both locally and remotely from in-memory data store as well as on-disk data store.

When a task completes, its output is sorted according to the key of each record and saved to local disk into a shuffle data file and an index file. This index file contains mapping from the children RDD’s partition id to the region of the data file that contain the data for that id. The locations of these files are then saved into the block manager of the executor. Only when this process is finished, can the task report back to the driver that it has been executed successfully.

After each task from of child RDD fetches the map output’s locations from the driver, it contacts the block managers specified in the locations to retrieve the data that belongs to its partition. For data that has already existed in the local block manager, it is read directly from the shuffle data and index file. On the other hand, data which need to be loaded from remote executors are buffered in-memory until the buffer is filled. The default buffer’s size is 48 MB. As the data from this buffer is consumed, new data will be fetched into the buffer, until everything is processed.
3.3 Extending Spark for coarse-grained pipelined stage execution

While changing Spark's runtime to use a fine-grained pipeline execution strategy, like in Storm or Flink, could give us a more contrasted comparison to the original batch execution strategy, it would be a very challenging task.

In the first place, it is impossible to fetch a task's output before that task finished entirely, because Spark's shuffle process is based on sorting. The output of a task need to be sorted and indexed before persisted to disk, so that the consumer tasks can fetch an entire region of shuffle file without having to do a full scan. In order to do a fine-grained pipeline shuffle, the shuffle module will need changing to a hash-base approach, so that each record can be forwarded directly to the correct consumer task based on the hash of its key.

In addition, in a fine-grained pipeline system, all consumer tasks have to be schedule to run concurrently with all producer tasks, otherwise the generated shuffle data cannot be pull by consumer from producer. In a Spark program, all stages are either producers or consumers, so all stages need to be scheduled to run concurrently. This will require a very large change to the scheduler.

Furthermore, one of the largest advantages of pipelining is the ability to reduce disk I/O by writing little to no data to disk. A pipeline shuffle, which write all data to disk, is self-defeating. However, Spark uses the on-disk shuffle files as part of its fault-tolerant mechanism. As a result of this conflict, a switch to a fine-grain pipeline shuffle also necessitates a new mechanism to recover from failure in computation.

For these reasons, a coarse-grained pipeline approach, inspired by [18], was chosen to evaluate in
this thesis. The shuffle component remains unchanged: the output of each task of the producer stage is still sorted and written to disk, and the tasks of the consumer stage still fetch data in large chunks instead of doing it record-by-record. However, Spark is modified so that the cluster does not wait for the producer stage to end before launching the consumer stage. On the contrary, some tasks of the consumer stage are started early to process the shuffle data even when they are incomplete. Furthermore, the consumer stage’s tasks are modified to wait for the currently running producer tasks when they have exhausted all available shuffle data.

This design allows the two stages to overlap, and takes advantage of the resource that would be wasted if the cluster waits for producer stage to end. Furthermore, because the shuffle data is still persisted to disk, the fault-tolerant mechanism of Spark is not compromised.

### 3.3.1 Early scheduling of the consumer stage

Since the consumer stage is dependent on the shuffle output of the producer stage, the tasks in the former will never finish until the later is done. Thus, we need to make sure that all the tasks in the parent stage are either already finished or currently running before we attempt to start the child stage. This requirement will guarantee that the child stage’s tasks will not compete for CPUs with the parent stage’s tasks and avoid the deadlock condition shown in figure 3.5.

In this figure, the cluster has 2 execution slots and can run 2 tasks concurrently, the parent and the child stage both have 4 tasks. An invalid scheduler assigns all 2 slots to the child stage, while the parent stage has only finished 3 tasks. Now these 2 running tasks are stuck because they do not have the output of the last pending tasks from the parent stage. Furthermore, this waiting task of the parent cannot be executed, because there are no free CPU slots. Thus, the program falls into a deadlock and cannot progress.

Figure 3.5: Invalid scheduler will cause deadlock when executing tasks. The red triangles indicates finished tasks, blue means waiting tasks, while green shows running tasks. The dash line represents dependencies between tasks of stage 2 to task of stage 1.

Figure 3.6 describes the correct scheduling behavior. For the same configuration of 2 execution slots
and 4 tasks for each stage, the child stage’s task are not scheduled until all tasks of the parent stage have either completed or are currently running. The output of the producer stage will soon be completed and ingested by the consumer stages.

![Figure 3.6: Correct barrier-less stage scheduling. The red triangles indicates finished tasks, blue means waiting tasks, while green shows running tasks. The dash line represents dependencies between tasks of stage 2 to task of stage 1.](image)

In order to implement this, we add a method into the CoarseGrainedSchedulerBackend to check if there are enough free CPU cores to execute all pending tasks. Whenever a task finishes, the DAGScheduler will use this method to check if it is safe to start the children stages, which is when the number of all pending tasks are less than the number of free cores. Once this condition is satisfied, the DAGScheduler scans the stages DAG and find all unexecuted stages whose parents are either finished or are already running, and starts them.

### 3.3.2 Processing incomplete shuffle data

In an unmodified version of Spark, when a task in the consumer stage starts executing, it contacts the MapOutputTracker to get the list of shuffle data’s locations that was produced by the producer stages. As all parents of this consumer stage have completed, the shuffle data have already been stored in executors’ block manager. Thus, the task can safely contact the block managers which house the shuffle data and retrieve it for processing.

However, this assumption no longer hold in our modified version of Spark, because the child stage can be started before the parents stages have finished. Consequently, the consumer’s tasks have to deal with incomplete shuffle data. In order to fulfill this requirement, we have to modify how the shuffle data location is saved and how it is read.

Firstly, the DAGScheduler is modified to update the MapOutputTrackerMaster with new shuffle location every time a task if finished. Previously, the location data is only published to the MapOutputTrackerMaster when the entire stage is completed. This change allows a consumer stage to always have some data to process when it is started, even if the producer stages have not finished.
Secondly, the **MapOutputTrackerWorker** in the executors is set to periodically update itself with new shuffle data locations from the driver, as long as the cached data contains "holes" - shuffle's partitions with empty location. This is done by running a dedicated **MapOutputUpdater** for each shuffle in a background thread. This updater is started when the first task in the consumer stage contacts the executor's **MapOutputTrackerWorker** for shuffle data, and continues running until the locations information of the shuffle is complete. Figure 3.7 shows the communication between the two **MapOutputTracker** in the executors and the driver. It begins as the executor sends the number of shuffle output locations it has to the driver, and asks the driver to return the new number of shuffle locations. If the driver already has more locations than the executor, it will response right away with the new number, which will trigger the executor to fetch the updated addresses. Otherwise, the executor’s request is put into a waiting list until a producer’s task is complete and the locations data is amended with this task’s shuffle output. This process repeats until the shuffle’s output locations and shuffle’s partitions are equal in number, at which point the **MapOutputUpdater** terminates itself.

![Figure 3.7: Map output trackers' communication](image)

Thirdly, changes has to be made to how the shuffle data is processed. Traditionally, Spark implements shuffle computation using the volcano iterator model [19] as shown in figure 3.8. The bottom layer, **ShuffleBlockFetcherIterator**, retrieves shuffle blocks from block manager locally or remotely, decodes, and exposes the data as an iterator of records. Each consumer’s task has its own iterator, and uses it to get records for processing.

We modify this design and add another layer, **PartialShuffleBlockFetcherIterator**, to handle missing data blocks. The **PartialShuffleBlockFetcherIterator** is started with a **ShuffleBlockFetcherIterator** seeded with all the available shuffle data at that moment. As soon as this initial iterator runs out, the **PartialShuffleBlockFetcherIterator** will contact the local **MapOutputTrackerWorker** to get the list of new shuffle blocks that have become available while the initial data was processed. Afterwards, a new **ShuffleBlockFetcherIterator** is created by the **PartialShuffleBlockFetcherIterator** to handle this new data. This process repeats until the producer stage completes and all shuffle data becomes available. The modified iterator model is presented in figure 3.9.
3.3.3 Over-scheduling executors

While doing our experiments, we noticed that during shuffle, the map output can be consumed a lot faster than it is produced. As a result, even when the consumer’s tasks are launched early, they spent a lot of time waiting for the shuffle data to be generated. Instead of letting the CPUs go to waste while the tasks are idle, we decided to launch additional tasks while the first few ones are waiting for more shuffle data. This is implemented by adding a hook inside the `PartialShuffleBlockFetcherIterator` to notify the executor when the initial shuffle data is completely processed and the task needs to block and wait for new data. On receiving this signal, the executor will attempt to launch an additional task, while the first task is waiting for more data. However, because memory requirement grows linearly with the number of tasks, launching too many tasks will cause the executor to quickly run out of memory. In order to avoid that situation, by defaults we do not launch more than one additional task for every early-start task running on an executor. In case this is still not restrictive enough, as shown later in one of our benchmark, this behavior can also be turned off completely by the developer of the application.

3.3.4 Fault-tolerance implications

As presented in section 2.2.1, Spark provides fault-tolerant computation by recalculating the RDDs based on their lineage. However, tracing the entire lineage from the first transformation can potentially take a lot of time, especially when data-shuffling is involved. As a result, Spark takes advantage of the on-disk shuffle files as a check-point mechanism. Because these files are never deleted until
the program finished, whenever a task in a stage need to be restarted due to some execution errors, they can re-read the input shuffle files to recompute the corresponding partition. This retry is performed transparently up to three times by the TaskScheduler, and we don’t have to make any modification to this module.

However, in case of catastrophic failure like the crash or disconnection of an executor process, the on-disk shuffle data will no longer be available. This error will trigger a `FetchFailedException`, when the currently running tasks try to load their shuffle input. In this case, the runtime cannot simply restart only the tasks that trigger the error, it has to recompute the missing shuffle data as well. In order to do this, the current design calls for the cancellation of the erroneous stage and the restart of its parents. Because there is only two layers of dependency: the parents stages and the children stages, when we cancel the children stages, the entire cluster will be free to run the parents stages.

On the other hand, when we remove the stage barrier using the considered pipelined data-transfer policy, we can have three or more layers of dependency: the parents, the children and the grandchildren.
For the purpose of clarity, we will called them $S_1$, $S_2$ and $S_3$, respectively. $S_1$, which is the parent of $S_2$, has already finished successfully. $S_2$ is the currently running stage, which is about to complete. $S_3$ is a child of $S_2$ which was started early to processed the shuffle data of $S_2$. In an unmodified Spark runtime, when $S_2$ encounters a FetchFailedException, it will be canceled and $S_1$ will be restarted as explained above. However, due to the existence of $S_3$, a deadlock can happen.

When $S_2$ is canceled, all its CPU slots are released so that the runtime can use them to run the tasks of $S_1$. Nonetheless, the runtime is also free to used these CPU slots to execute the tasks of $S_3$. The deadlock happens when all CPU slots are allocated to $S_3$ instead of $S_1$. Because $S_2$ have not finished, $S_3$'s will be stuck waiting for addition data from $S_2$. However, $S_2$ need the result of $S_1$, which cannot be executed due to the lack of CPU slots.

This deadlock is quite likely to happen, because there is a timeout between the time when $S_2$ is canceled and the time $S_1$ is rescheduled. This delay is an optimization to handle multiple FetchFailedExceptions which arrive in quick succession with a single restart command. During this period and the time spent on the delayed scheduling algorithm mentioned in section 3.2.2, $S_3$ might get to occupy all the free CPU slots and cause the starvation of $S_1$.

In order to avoid this deadlock, whenever $S_2$ encounters an FetchFailedException error and have to be canceled, we also cancel $S_3$. This action free up the cluster resources to restart $S_1$. This is the closest behavior to the original version of Spark. However, it also means that the CPU time spent on the $S_3$ is wasted. Unlike $S_2$, who gets to keep the result of the tasks which have finished before the error, $S_3$ have no successful tasks and have to be restart entirely. This, however, can be avoided with a more intelligent scheduler. Because only the faulty parts of $S_1$ and $S_2$ need to be recomputed, it should not be necessary to fully cancel $S_3$. We only need to cancel a small number of tasks in $S_3$, so that we have enough CPU slots to restart $S_1$ and $S_2$. Unfortunately, this is difficult to implement with the current scheduler’s architecture of Spark, so canceling $S_3$ completely is our best choice.
Chapter 4

Evaluation & Discussion

4.1 Environment

4.1.1 Hardware

This evaluation is performed on the INESC-ID cluster. All machines have 2 Intel Xeon 5506 CPUs with 4 cores each, and 48 GB of RAM and are connected using 1 Gbps ethernet. However, the experiments are not run directly on the physical machines, but on OpenStack’s virtual machines. The virtual machines are configured to use 4 CPU cores each and 16 GB or RAM each, which is similar to the size of an Amazon EC2 m4.xlarge VM [20] at the time of writing. All evaluation is done using 5 physical machine, each hosting only 1 VM. One virtual machine is used as the Spark’s master and HDFS’s namenode, while the other four are used to run Spark’s workers and HDFS’s datanodes.

4.1.2 Software

We implemented our modifications using Spark version 1.6.2. Other software used in the benchmark includes:

- **OpenStack**: This is used to launch and manage virtual machines on top of the physical machine.
- **Ubuntu Linux 14.04**: This is the operating system used in the virtual machine
- **HDFS 2.7.2**: Hadoop Distributed File System is used to stored input data and output data
- **Java 7**: Spark and Hadoop are run on Java 7, which is the latest version available for Ubuntu Linux 14.04
- **Ansible 2**: We use Ansible to manage the deployment of each virtual machine.

All virtual machines have identical hardware and software configuration.
4.2 Methodology

In order to evaluate the performance difference when changing Spark's execution strategy from batch to pipeline, multiple benchmarks were performed with a variety of configurations, which will be presented below. For each benchmark and configuration, we performed the following procedure:

1. Execute the benchmark multiple times on with and without our modification to Spark to support pipelining.

2. The execution time of each run is collected and grouped into two sets based on whether it makes use of our modification or not.

3. These two sets are sorted and filtered to remove the top and bottom 10%. This step gets rid of the extreme value that could skew the result.

4. Calculate the mean, standard deviation and the 95% confidence interval of the execution time for each set. In addition, we also calculated the mean difference of the two sets to quantify our improvement.

5. Perform the unequal variance t-test (Welch’s t-test) [21] on the two sample sets with the null hypothesis that the two population means are equal. If this hypothesis can be rejected with 95% confident, then we can conclude that overlapping stage execution causes a statistically meaningful change in execution time.

All benchmark applications were configured so that the number of partitions in a RDD is double the total number of CPU cores in the cluster, as recommended in Spark’s developer manual. In addition, we allow over-subscribing executors in all but the Triangle Count benchmark in section 4.5.3, because the Triangle Count benchmark would not complete with that option enabled.

4.3 Micro-benchmark: Randomize Text

4.3.1 Description

We start our experimental evaluation by presenting the results achieved while using a micro-benchmark which allowed to stress, in a controlled manner, the performance of the inter-stage data-transfer mechanism. We developed a micro-benchmark to determine the effects of switching Spark’s execution strategy from batch to pipeline in the ideal conditions. The requirements for this benchmark are being both computation and I/O intensive and have an adjustable number of stages. After much testing, we settled on a benchmark that works as followed:

1. Read a text file into a RDD, where each record is a line in the input file. This RDD is materialized and kept in memory to use in step 2. The reading time was recorded and subtracted from the result because it was not a part of the shuffle process and was the same whether running on our modified version of Spark or not.
2. Perform a random shuffle of the characters within each record in the RDD. This step aims to satisfy the computation-intensive condition.

3. Re-partition the resulting RDD in the previous step. Due to the random shuffle, most records will need to be transfer to a different computer, making this workload I/O intensive.

4. Repeat step 2 and 3 for as many times as necessary to achieve the required number of stages.

The experiments were performed with the input size of 0.6 GB, 1.2 GB and 1.8 GB and the number of iterations set to 4, 8 and 16.

4.3.2 Result & Discussion

Figure 4.1 describes the time during which an executor is actively running a task for the workload of 0.6 GB input and 4 iterations. The red bar indicates the entire duration of a stage, while each of the four other bars represents the duration of each executor. The gray area in a executor’s bar signals that some tasks in that executor have to wait for the data produced in a preceding stage.

In figure 4.1a, which reports the results obtained using the unmodified Spark, the executors always start together at the beginning of the stage, and a slow executor will force the others to wait for it. This limitation is removed when running the application when using the pipelined data transfer scheme: the executors can be assigned new task as soon as possible, and the stages are nicely overlapped as shown in figure 4.1b

![Figure 4.1: Timeline of Active Executors for Randomize Text](image)

Figure 4.2, 4.3 and 4.4, show the difference between execution time on our platform and the original Spark, when the input size is changed between 0.6 GB, 1.2 GB and 1.8 GB. As the amount of data needs to be shuffled increased, the cost of I/O operation overshadowed the cost of CPU operation. Because overlapping stages focuses on exploiting the free CPU time at the end of each stage, the improvement became less significant when you raise the size of data. Nevertheless, we still managed to achieve from 15 - 20 % of speed up in most tests of this benchmark.
Figure 4.2: Comparison of Randomize Text Execution Time - 4 Iterations

Figure 4.3: Comparison of Randomize Text Execution Time - 8 Iterations

Figure 4.4: Comparison of Randomize Text Execution Time - 16 Iterations
Figure 4.5, 4.6 and 4.7 indicate a decrease in the relative execution time between running the application using our pipeline implementation and the original batch strategy, as the number of iterations increase. These results are in line with our expectation, because the number of iterations is tied directly to the number of stages in this benchmark. In addition, the more stages we have in our application, the higher performance we can extract from the cluster by overlapping them. However, the performance did not improve linearly with the number of stages. In fact, with the same input size, doubling and quadrupling the number of iterations brought very little change in relative runtime and peaked out at 22%. This can be explained by considering the fact that the use of a given inter-stage data transfer policy do not affect the performance of the first stage of execution of a job (which has no dependency on previous stages). By increasing the number of stages, the relative impact on overall execution time of the first stage tends to become less and less relevant.

Figure 4.5: Comparison of Randomize Text Execution Time - 0.6 GB

Figure 4.6: Comparison of Randomize Text Execution Time - 1.2 GB
Since the differences are so large between Spark runtime and our runtime, we fully expected that all the tests for this benchmark will pass the t-test. This expectation is confirmed with the detailed statistic in table 4.1.

<table>
<thead>
<tr>
<th>Input</th>
<th>Iteration</th>
<th>Mean</th>
<th>SDev</th>
<th>No of Tests</th>
<th>Mean</th>
<th>SDev</th>
<th>No of Tests</th>
<th>Speedup</th>
<th>Orig = Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.6 GB</td>
<td>4</td>
<td>92.4</td>
<td>3.1</td>
<td>33</td>
<td>75.1</td>
<td>2.3</td>
<td>33</td>
<td>18.7 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>0.6 GB</td>
<td>8</td>
<td>180.8</td>
<td>4.9</td>
<td>33</td>
<td>144.6</td>
<td>3.3</td>
<td>33</td>
<td>20 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>0.6 GB</td>
<td>16</td>
<td>362</td>
<td>7.1</td>
<td>33</td>
<td>282</td>
<td>4.0</td>
<td>33</td>
<td>22.1 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.2 GB</td>
<td>4</td>
<td>157.5</td>
<td>3.1</td>
<td>33</td>
<td>135.4</td>
<td>4.6</td>
<td>33</td>
<td>14.0 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.2 GB</td>
<td>8</td>
<td>311.5</td>
<td>4.3</td>
<td>33</td>
<td>263.3</td>
<td>4.3</td>
<td>33</td>
<td>15.5 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.2 GB</td>
<td>16</td>
<td>636.9</td>
<td>9.3</td>
<td>33</td>
<td>535.8</td>
<td>9.2</td>
<td>33</td>
<td>15.9 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.8 GB</td>
<td>4</td>
<td>222.4</td>
<td>4.2</td>
<td>33</td>
<td>196.1</td>
<td>5.2</td>
<td>33</td>
<td>11.8 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.8 GB</td>
<td>8</td>
<td>446.8</td>
<td>4.4</td>
<td>33</td>
<td>380.7</td>
<td>10.1</td>
<td>33</td>
<td>14.8 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>1.8 GB</td>
<td>16</td>
<td>869.9</td>
<td>7.7</td>
<td>33</td>
<td>745.7</td>
<td>10.9</td>
<td>33</td>
<td>14.3 %</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

Table 4.1: Randomize Text Execution Statistics

Figure 4.8, shows the timeline for each stage of the randomize text application, it clearly illustrates that the effect of overlapping stages become more noticeable as the number of stages increases.

Overall, the results obtained using our micro-benchmark show that in workloads that make intensive use of operators requiring inter-stage data transfers, a batch-based policy, such as the one currently adopted by Spark, can achieve performance that is up to 22% lower than a pipelined approach.
Figure 4.8: Randomize Text Execution Timeline for 0.6 GB Input
4.4 Classic two-stage cluster processing benchmark

Two commonly used benchmarks to evaluate classic Map/Reduce-like frameworks: Words Count and Terasort, are tested in this section. These workloads have the same characteristic of using only two stages.

4.4.1 Words Count

Description

Words Count is an application that counts the number of occurrence of each distinct word in a document. The algorithm consists of two stages:

1. The document is loaded and split into words, each word is then transformed to a tuple that has the form \((\text{word}, 1)\).

2. The tuples are partitioned so that all tuple with the same word are grouped together in the same data partition.

3. Merge all tuples that have the same word and add the counter together.

We randomly generated 3 text documents with the size of 3.2 GB, 6.4 GB and 9.6 GB to use as the input for the Words Count application.

Result & Discussion

Figure 4.9 describes the timeline of each executors when running the Words Count application on our modified Spark. Although stage 1 can be started early, the executors spent the majority of the time waiting for the shuffle data from stage 0, as indicated by the large gray area in each executor’s bar.

![Figure 4.9: Timeline of Active Executors on modified Spark for Words Count](image-url)
Figure 4.10 shows a comparison of execution time between running **Words Count** using the original batch approach and our modified pipelined approach. Although running **Words Count** with our modification to Spark did give a reduction in the program’s duration according to this result, the difference is minimal and not noticeable. This was confirmed by the statistic test result in table 4.2. When performing the t-test on the samples, we were not able to reject the null hypothesis with 95% confident.

![Figure 4.10: Comparison of Words Count Execution Time](image)

<table>
<thead>
<tr>
<th>Input size (gigabytes)</th>
<th>Batch</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2 GB</td>
<td>Mean 35.1</td>
<td>35.1</td>
</tr>
<tr>
<td></td>
<td>SDev 0.7</td>
<td>0.7</td>
</tr>
<tr>
<td></td>
<td>No of Tests 33</td>
<td>33</td>
</tr>
<tr>
<td>6.4 GB</td>
<td>Mean 67.9</td>
<td>67.9</td>
</tr>
<tr>
<td></td>
<td>SDev 2.2</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>No of Tests 33</td>
<td>33</td>
</tr>
<tr>
<td>9.6 GB</td>
<td>Mean 94.8</td>
<td>94.8</td>
</tr>
<tr>
<td></td>
<td>SDev 2.1</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>No of Tests 33</td>
<td>33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Speedup</th>
<th>Orig = Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9 %</td>
<td>Not Rejected</td>
</tr>
<tr>
<td>0.5 %</td>
<td>Not Rejected</td>
</tr>
<tr>
<td>1.1 %</td>
<td>Not Rejected</td>
</tr>
</tbody>
</table>

This lack of difference can be explained by investigating the timeline of each stages, shown in figure 4.11. While overlapping stages focuses on speeding up the second stage, the majority of execution time is spent on the first stage with the **Words Count** workload. Furthermore, the fact that there is no performance regression, indicates our modification to support pipelining has negligible overhead.

### 4.4.2 Terasort

**Description**

*Terasort* is a benchmark to sort one Tera byte of randomly distributed data. When implemented in Spark, it consists of two stages:

1. The input data is read and divided using a range partitioner. For example, in order to have \( n \) partitions, we need to choose \( n \) number \( a_1, a_2, ..., a_n \) so that \( a_1 < a_2 < ... < a_n \). Then the number \( i \) will belong to partition \( j \) if \( a_j \leq i < a_{j+1} \)

2. Sort each partition and output the data
Due to limited resource, we were not able to test Terasort with the full 1 TB of input, which would require hundreds of machines to finish in a reasonable amount of time. Instead, we generated 3 input sets with size 1.6 GB, 3.2 GB and 6.4 GB. These sizes allowed us to repeat the application many times and collected the necessary number of samples for the statistic tests.

**Result & Discussion**

Similar to Words Count, when the stages are overlapped, the executors spend a significant amount of time waiting for shuffle data in the Terasort applications, as shown in figure 4.12. In this sample, some stage-0 tasks running in the blue executor were lagging behind and the eagerly-started tasks in stage 1 had to wait for them. As soon as stage 0 finished, all tasks stopped waiting.

As illustrated in Figure 4.13, running Terasort using the pipeline strategy showed only a minor improvement over the original batch strategy of Spark at the best scenario. In the case of 1.6 GB input,
there is even a small performance regression of 1.89%. Nevertheless, the difference between the two configuration is so small that it is not possible to reject the null hypothesis with 95% confident, as shown in table 4.3.

![Terasort Absolute Execution Time](image1)

(a) Absolute value

![Terasort Relative Execution Time](image2)

(b) Relative value

Figure 4.13: Comparison of Terasort Execution Time

<table>
<thead>
<tr>
<th>Batch Pipeline</th>
<th>Graph</th>
<th>Mean</th>
<th>SDev</th>
<th>No of Tests</th>
<th>Mean</th>
<th>SDev</th>
<th>No of Tests</th>
<th>Speedup</th>
<th>Orig = Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.6 GB</td>
<td></td>
<td>45.3</td>
<td>9.1</td>
<td>33</td>
<td>46.1</td>
<td>8.5</td>
<td>33</td>
<td>1.9 %</td>
<td>Not Rejected</td>
</tr>
<tr>
<td>3.2 GB</td>
<td></td>
<td>148.4</td>
<td>20.6</td>
<td>33</td>
<td>147.3</td>
<td>19.1</td>
<td>33</td>
<td>0.7 %</td>
<td>Not Rejected</td>
</tr>
<tr>
<td>6.4 GB</td>
<td></td>
<td>482.4</td>
<td>18.3</td>
<td>33</td>
<td>477.4</td>
<td>18</td>
<td>33</td>
<td>1.0 %</td>
<td>Not Rejected</td>
</tr>
</tbody>
</table>

Table 4.3: Terasort’s execution statistics

As seen in Figure 4.14, with only two stages, there are not a lot of chance for enhancement. Overlapping the two stage of the TeraSort application only gives an unnoticeable 1% decrease in execution time when input size is 6.4 GB during our testing.

To sum up, changing from batch execution to pipeline execution in Spark does not seem to affect the performance of the classic two-stage Map/Reduce-like applications. Fortunately, thanks to Spark powerful API, developers rarely have to write these type of applications. Instead, they are encouraged to combine multiple traditional two-stage applications into a single multi-stage application to take advantage of Spark’s pipelining and in-memory caching capability. These type of application will be tested in the next section.
4.5 Multi-stages benchmarks

In this section, we evaluate three graph algorithms: PageRank, Connected Components, and Triangles Count. All three algorithms have multiple stages and are implemented in the GraphX library, which is provided along with Spark. In order to focus on the actual processing time of the data analytic jobs, we removed the time it took to load the graph into memory from the graph loader, because it was not a part of the algorithm and keeping it skew the results.

4.5.1 PageRank

Description

PageRank [22] was the original algorithm used by Google to rank website in their search engine results. In this algorithm, the Internet is a graph where each website is represented by a vertex and a link from website A to website B is represented by an edge from vertex A to vertex B. Each website has a ranking, calculated based upon the ranking of other websites that link to it.

PageRank is an iterative algorithm with many stages, so it’s naturally a good candidate for improvement by breaking the stage barrier. We tested the PageRank application with multiple random, artificial graphs of different sizes, whose vertex’s number of outgoing edges (out degrees) follow the log-normal distribution:

\[ P(d) = \frac{1}{d\sigma\sqrt{2\pi}} e^{-\frac{(\ln d - \mu)^2}{2\sigma^2}} \]

where \( \sigma = 1.3 \) and \( \mu = 4 \). According to Malewitz et al. in [23], this is a good approximation for many types of real-world graphs.

In our benchmark, we generated four graphs with 1, 2, 3 and 4 millions vertices to run PageRank on. We decided on these sizes because we wanted to repeat the experiment a lot of times, which would
take too long if the graphs are overly large. Each execution of the experiment was run using a 3-iteration version of PageRank and the total times minus the time to read the input graph was recorded.

Result & Discussion

In the PagePank application, many early-started tasks are also forced to wait for shuffle data. This is most prominently shown by the gray area in all executors during stage 8 and 20, in figure 4.15.

![Figure 4.15: Timeline of Active Executors on modified Spark for PageRank](image)

Figure 4.16 shows the difference in execution time of the PageRank algorithm between the pipeline approach in our modification and the original batch approach used in Spark. We can see that the gap between our version of Spark and the original increases with the size of the graph. With the 1-million graph, the improvement is only about 5.7 %, while with the 3-million graph, this has increase to around 16.6 %. However, when the graph size reach 4 million, the improvement decrease slightly to 15.3 %. This differs from the results in the Randomize Text benchmark, due to the characteristic of the PageRank workload: high CPU, low I/O usage.

Table 4.4 shows the raw statistics and results of the t-test. As indicated, the t-test succeeded in
rejecting the null hypothesis with 95% confidence. Thus, we are quite confident that this result is statistically significant.

Table 4.4: PageRank Execution Statistics

<table>
<thead>
<tr>
<th>Graph Size (vertices)</th>
<th>Batch</th>
<th>Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph</td>
<td>Mean</td>
</tr>
<tr>
<td>1M</td>
<td></td>
<td>64.7</td>
</tr>
<tr>
<td>2M</td>
<td></td>
<td>142.2</td>
</tr>
<tr>
<td>3M</td>
<td></td>
<td>248.1</td>
</tr>
<tr>
<td>4M</td>
<td></td>
<td>508.9</td>
</tr>
</tbody>
</table>

Figure 4.17 breaks down the detail execution time for each stage in our PageRank implementation when running with the 3-million vertices graph. The majority of the program's duration is spent on three jobs, one for each iteration of PageRank. The first job contains some addition transformations to initialize the data, and have no caches, so it is longer than the next two. It is clear that running the algorithm using our modified Spark allows the stages to be overlapped from the beginning to the end of each job and decreases the overall execution time.

### 4.5.2 Connected Components

**Description**

A connected component [24] in a graph is a sub-graph in which any two vertices are connected to each other, either directly or through other vertices. In other word, starting from one vertex, you can get to any other vertices in the same connected component just by following the edges.

Spark's implementation of the Connected Components algorithm run iteratively in multiple super steps. First, the value of a vertex is initialized to its ID. Then, in each step, each vertex will send its value to its neighbors and also receive messages in return. When a vertex receives a message, it will compare its current value with the value in the message, and set its value to the smaller of the two. The program terminates when all vertices stop changing value.
Figure 4.17: PageRank's Execution Timeline for the 3 millions vertices graph

The same graphs, which served as input for the PageRank benchmark, were reused to evaluate the connected components program.

**Result & Discussion**

Connected Components’ timeline for active executors in figure 4.18 looks quite similar to the PageRank’s one in figure 4.15: both consist of multiple jobs, with multiple stages in each job. There are also the appearance of the gray areas in multiple executors, which indicates that the shuffle data was consumed by child stage faster than they are produced by parent stages. In addition, one can clearly see that the three stages 18, 41, and 77 dominate the execution time of the three jobs 2, 3, and 4, respectively.

In Figure 4.19, we shows the difference between execution time when running the Connected Components algorithm using pipeline execution and using batch execution. The difference was not as large as in our micro-benchmark, or in the PageRank application, ranging from only 2.4 % to 5.6 %. There are two reasons for this result.

- The first reason is the Connect Components programs contains many smaller jobs, each job requires an implicit barrier at the end to in order to facilitate data caching. Furthermore, our modified Spark has no ways to overlap stages across a job barrier, because the driver program have to wait for the current job to complete before scheduling the next job. This limitation reduces the effectiveness of our platform for this workload.

- The second reason is that most of the time in a job was spent on a single large stage. As previously seen in the Words Count application, if the size of two overlapping stage are too different, the time saving will be minimal.

Table 4.5 contains more detailed statistics data and the t-test result. As with PageRank, we can reject the null hypothesis that our modification has no influence, with 95 % confident in all cases.
Figure 4.18: Timeline of Active Executors on modified Spark for Connected Components

Figure 4.20 describes the timeline for each stage in the Connected Components algorithm when running on the graph with 3 million vertices. The program finished in 5 super steps, represented by the 5 jobs shown in the chart. Again, running the algorithm on our modified version of Spark show up to 5 % improvement over the original.

4.5.3 Triangles Count

Description

The Triangles Count algorithm determines the number of triangle to which a vertex belongs, for all vertices in the graph. A vertex is considered part of a triangle if it has two neighboring vertices with an edge between them. This algorithm is chosen for testing because it is very memory intensive and generates a large amount of shuffle data.

The Triangles Count algorithm consists of three steps:

1. Compute the set of neighbors for each vertex.
2. For each edge, calculate the intersection of the source’s neighbors set and the destination’s neighbors set. The size of the intersection is the number of triangle that include both the source, the destination and one item in the set. Send this value to both the source and the destination vertices.

3. Sum up the received values and divide by 2, because each triangle is counted twice in the previous step.

Due to the large memory requirement of the Triangles Count algorithm, we were not able to run it with the graph used for PageRank and Connected Components benchmarks. Instead, we generated four new graphs of size 100 thousands, 200 thousands, 300 thousands and 400 thousands vertices to perform this test.

**Result & Discussion**

With the default settings, the application crashed with memory errors when running on our modified Spark with the 2 largest graphs. To make the application complete successfully and to keep the same settings between all tests, we decided to disable the executors over-scheduling functionality for all Triangles Count benchmarks.

Upon investigation, we figured out that the reasons for this error is due to the launches of addition tasks when we over-scheduling the executors. Although an early-started task waits for shuffle data, it will consume negligible CPU time, but can still require the as much memory as a running tasks. When over-scheduling executors is enabled, we launch a second task for every waiting tasks, which doubles the memory requirement. In memory-hungry workload, a lot of time might be spent doing garbage collection.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Mean</th>
<th>SDev</th>
<th>No of Tests</th>
<th>Speedup</th>
<th>Orig = Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>1M</td>
<td>82.5</td>
<td>2.3</td>
<td>30</td>
<td></td>
<td>Rejected</td>
</tr>
<tr>
<td>2M</td>
<td>170.6</td>
<td>11.1</td>
<td>31</td>
<td>2.4 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>3M</td>
<td>332.2</td>
<td>9.3</td>
<td>30</td>
<td>5.6 %</td>
<td>Rejected</td>
</tr>
</tbody>
</table>

Table 4.5: Connected Components Execution Statistics
collection when this happens. In the worst scenario, the executor will exit because it runs out of memory, causing the entire application to fail.

In figure 4.21, which displays the active time of each executor while running Triangles Count program, it is clear that the gray areas of stage 7 appear and disappear in small interval. This is because the result of each running tasks in stage 6 is immediately made available to the early-started tasks in stage 7, which unblock them to continue processing.

The graph 4.22 shows the average execution time of the Triangle Counting program. For the 100 thousands vertices graph, running the application with pipeline execution brings an improvement of 20.6 %, close to the largest reduction in execution time we had in the randomize text benchmark. However, in the test with bigger graph the result is not so good: only the 300 thousands vertices graph show a minor enhancement of 5.5 %, while the other two test show no noticeable difference. The only explanation we have for these widely different results is due to memory-pressure. The Triangles Count algorithm creates some very heavy memory-hungry stage. When a consumer stage is launched, it competes for RAM with the still running producer stages. For a small graph, there is no contention because there is enough memory to run both the producer and the consumer, but for larger graphs this side-effect can slow down the producer stages significantly.
Figure 4.21: Timeline of Active Executors on modified Spark for Triangles Count

Figure 4.22: Comparison of Triangles Count Execution Time
Table 4.6 displays the detailed statistics data and the t-test result. Despite we have a small performance regression in the 200 thousands vertices graph, we are not able to reject the null hypothesis with confidence 95 % for this case. This means that the different between our modified Spark and the original Spark are, in fact, not distinguishable under this condition.

One issue we encountered with the Triangles Count workload is a noticeable contention between the stage before the parent barrier and the child stage. Due to the way Spark schedule tasks into machines, some tasks from the child stage can be launched in the same computer that is still running tasks from the parent stages. This co-location can lead to further reduction in speed of the slowest tasks from the parent stages, which undesirably increases the parents stage execution time. In figure 4.23a, which compare the duration of each stage between the two testing configuration, stage 6 and 7 finished later when running on our modification. This increase in stages’ duration explains the poor results. Similarly, when the graph size is 300K in figure 4.23b, but in this case, the overlapping of stage 6 and 7 was enough to overcome the increase in the duration of stage 6.

In summary, switching to pipeline execution was able to enhance the performance of multi-stage workloads further than traditional 2-stage Map/Reduce-like workloads. However, the amount of improvement varied greatly depending on the application. For PageRank, we witnessed as much as 15 % decrease in the algorithm’s total execution time. When processing small graph, the Triangles Count workload was able to achieve 20 % performance increase; however, when the size of the graph increase, the performance difference between original Spark and our modified runtime dropped to almost zero. Finally, the Connected Components benchmark stayed in the middle, giving a minor 2 - 5 % reduction in the program’s running time for all tested graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Batch Mean</th>
<th>Batch SDev</th>
<th>Batch No of Tests</th>
<th>Pipeline Mean</th>
<th>Pipeline SDev</th>
<th>Pipeline No of Tests</th>
<th>Speedup</th>
<th>Orig = Mod</th>
</tr>
</thead>
<tbody>
<tr>
<td>100K</td>
<td>64.2</td>
<td>6.4</td>
<td>30</td>
<td>51.0</td>
<td>3.3</td>
<td>30</td>
<td>20.6 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>200K</td>
<td>161.2</td>
<td>20.0</td>
<td>30</td>
<td>164.3</td>
<td>20.6</td>
<td>30</td>
<td>-1.9 %</td>
<td>Not Rejected</td>
</tr>
<tr>
<td>300K</td>
<td>318.9</td>
<td>27.9</td>
<td>30</td>
<td>301.4</td>
<td>26.8</td>
<td>30</td>
<td>5.5 %</td>
<td>Rejected</td>
</tr>
<tr>
<td>400K</td>
<td>717.9</td>
<td>44.1</td>
<td>33</td>
<td>718.4</td>
<td>52.7</td>
<td>33</td>
<td>-0.1 %</td>
<td>Not Rejected</td>
</tr>
</tbody>
</table>
Figure 4.23: Triangles Count Execution Timeline
Chapter 5

Conclusions

Apache Spark is a very promising cluster processing framework with advance features that simplify the development of massive data processing application. It is, however, not without flaw. Because Spark uses a batch execution model to perform data shuffling, it requires a synchronization barrier in order to generate the batch. All computers in the cluster have to reach this barrier before the execution can continue, but due to the many factors that influence each machine, they can arrive at the barrier at widely different times. This heterogeneity means the computers that run faster will have to wait idly for the slower ones, which causes a waste of cluster resources. In addition, since a Spark program can contain many operators and thus many data shuffling operations, the idle CPU time can accumulate and become quite significant.

In this thesis, I have modified Spark to include a coarse-grained pipeline execution model. This modification is done by allowing two consecutive stages to overlap and data transfer between them is pipelined at task-level granularity. The tasks from the stage, which need to consume the data, are allowed to progress as long as they still have input data. However, they will wait for the rest of the result of the data-producing stages if these data are still not available after all other data has been processed. This guarantee enables us to run any Spark applications without any modification, which greatly increase the usefulness of our approach.

Furthermore, an extensive evaluation have been performed to evaluate the performance difference between the batch and the pipeline model in Spark. When the workload includes many shuffle operations, we have shown that a 10 to 20 % reduction in execution time is achievable, without having to change a single line of application code. In other less favorable workloads, we still saw a modest 5 % drop in the program’s runtime. Nevertheless, we still have workloads, such as Words Count and Tera-sort, where we detected no statistically significant difference in performance between the batch and the pipeline execution model. The most significant problem only happens when we were too aggressive with running the consumer stages, which crashed the application in some Triangles Count workloads. Consequently, we added a configuration to reduce this aggressiveness to allow the application to complete successfully. In practice, most big-data analytics application require tuning for maximum efficiency, so adding another configuration will not be too much trouble.
To conclude, the result of our experiments indicates that switching from a batch execution model to a pipeline execution model seems to have a positive result in the performance of Spark. However, the experiments also showed that the effectiveness of pipelining heavily depends on the characteristics of the applications.

5.1 Future work

5.1.1 Fine-grained pipeline shuffle

Even with the coarse-grained pipeline strategy evaluated in this thesis, Spark still need puts shuffle data to disk then fetches them in the next stage, and have to pay the heavy price of disk I/O. It would be quite beneficial if the disk I/O can be avoided by switching to a fine-grained pipeline shuffle that transfers records directly between the operators. Nevertheless, as discussed in section 3.3, this would require a major engineering effort and possibly the sacrifice of Spark's current intra-job fault-tolerant mechanism.

5.1.2 Memory-aware scheduler

The scheduler of Spark currently only takes into account of free CPU cores when it decides where to launch a task. This limitation can lead to trouble if a program uses a lot of memory. For example, while an executor might have 4 free CPU cores, it might not have enough memory to run 4 tasks at the same time. In fact, we encounter 2 issues during evaluation that are directly related to this restriction. An enhancement could be made to Spark's scheduler to determine if it is safe to launch a task based on the amount of memory available in the executor and the amount needed to finish a task.

Implementing this changes would be easy if the memory requirement of a task could be accurately estimated prior to execution. However, this job is very difficult when a task is fully customizable by the users. In a more limited environment, such as in SparkSQL, where the tasks are generated from SQL query by the runtime, it should be possible to reliably determine the resource usage of a task and use that information to schedule the tasks optimally.

5.1.3 Support other types of cluster manager

Spark supports many cluster managers: Yarn, Meso, and its own standalone implementation. The modification, described in this documents, relies on some information provided by the standalone cluster manager, which might or might not be available on Meso or Yarn.

In addition, the standalone cluster manager assume that the system is exclusively used for Spark. On the contrary, Yarn and Meso assume that the system can be shared between Spark and other frameworks, which will have implications on the availability of executors. Further investigations will be needed to determine whether this modification is applicable to Yarn, Meso or other type of cluster managers that Spark will support in the future.
Bibliography


