Parallelization with Standard ML

A Standard ML API for Hadoop and comparison with existing large-scale solutions

Master's Thesis in Computer Science

Ngoc Tuan Nguyen

May 14, 2015
Halden, Norway

Østfold University College www.hiof.no

Abstract

In recent years, the world has witnessed an exponential growth and availability of data. The term "big data" has become one of the hottest topics which attracts the investment of a lot of people from research to business, from small to large organizations. People have data and want to get insights from them. This leads to the demand of a parallel processing models that are scalable and stable. There are several choices such as Hadoop and its variants and Message Passing Interface. Standard ML is a functional programming language which is mainly used in teaching and research. However, there is not much support for this language, especially in parallel model. Therefore, in this thesis, we develop a Standard ML API for Hadoop called *MLoop* to provide SML developers a framework to program with MapReduce paradigm in Hadoop. This library is an extension of Hadoop Pipes to support SML instead of C++. The thesis also conducts experiments to evaluate and compare proposed library with other notable large-scale parallel solutions. The results show that *MLoop* achieves better performance compared with Hadoop Streaming which is an extension provided by Hadoop to support programming languages other than Java. Although its performance is really not as good as the native Hadoop, *MLoop* often gets at least 80% the performance of the native Hadoop. In some cases (the summation problem for example), when strengths of SML are utilized, *MLoop* even outperforms the native Hadoop. Besides that, *MLoop* also inherits characteristics of Hadoop such as scalability and fault tolerance. However, the current implementation of *MLoop* suffers from several shortcomings such as it does not support job chaining and global counter. Finally, the thesis also provides several useful guide-lines to make it easier to choose the suitable solution for the actual large-scale problems.

Keywords: Standard ML, Hadoop, MPI, MapReduce, Evaluation, large-scale, big data, parallel processing.

Acknowledgments

I have relied on many people directly and indirectly to finish this thesis. First of all, I would like to thank my supervisor Roland Olsson at Østfold University College. He guided me in the work with this thesis. Thank you for your great support.

I would also like to thank professor Øystein Haugen for your invaluable effort, proof-reading and very useful discussion at the end of my thesis.

I am particularly grateful to my family, my friends, especially my parents, for your continued support during my studying period in Norway. Your encouragement of both mental and material aspects is a big motivation for me to finish this thesis.

Prerequisites

This thesis covers many technologies in large-scale processing as well as relates to different programming languages. It is impossible to go into detail on every aspect that is mentioned in this thesis. Therefore, it is assumed that the reader has a background knowledge of computer programming. The background chapters of related technologies only introduce most important aspects.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

We are living in the Digital Era: information is instantly and readily available to more people than ever before. People take pictures on their phone, upload videos, update social network status, surf the web and much more. Data are being generated with an incredible speed. According to a study in 2014 [5], every minute of the day, we send 138 million emails, generate 4.7 million posts on Tumblr; YouTube users view over 5 million videos; Google processes 2.66 million searches. Therefore, "big data" becomes a fact of the world and certainly an issue that real-world systems must cope with. Furthermore, there is a claim that "more data lead to better algorithms and systems for solving real-world problems" [25]. Indeed, Banko and Bill [12] published a classic paper discussing the effects of training data size on classification accuracy in natural language processing. They concluded that several classification algorithms got better accuracy with more data. This leads to a consideration between working on algorithms and gathering data. With increasing amounts of data, a "good-enough" algorithm can provide very good accuracy.

In recent years, there has been a growing demand for the analysis of user behavior data. Electronic commerce websites need to record user activities and analyze them to provide users useful and personalized information. However, gathering user behavior data generates a massive amount of data which many organizations cannot handle in terms of both storage and processing capacity. A simple approach is to discard some data or only store data for a certain period. This can lead to lost opportunities because useful value can be derived from mining such data. As a result, there is a very high demand for storing and processing a very big amount of data. This task currently becomes a challenge for lots of companies like Google, Facebook, Yahoo, Microsoft. As a leader in many aspects of technology, in 2004, Google introduced MapReduce [15] - a programming model for processing and generating large datasets. This paradigm has drawn lots of attention from open-source communities and even big organizations. An open source implementation of MapReduce was developed, which called Hadoop. It was created by Doug Cutting and Mike Cafarela, then got support from Yahoo and now becomes an Apache project. For years, MapReduce and its implementations (Hadoop for example) become a great solution for lots of companies and organizations to cope with big data.

1.1 Related Work

The urgent demand of processing massive datasets have inspired lots of efforts this field. Hadoop is the solution that many people take into account to solve their problems. Therefore, lots of work have been done by researchers to improve Hadoop's performance in different aspects.

1.1.1 Improving Hadoop

Zaharia et. al. suggested a new scheduler for Hadoop system to improve the performance [34]. In their work, they listed assumptions in Hadoop's scheduler and pointed out how these assumptions were break down in reality. For example, assumption "starting speculative task on an idle node cost nothing" is break down when resources are shared. In this case, network bottleneck, disk I/O competition can happen. From the judgment, they proposed a new scheduler for Hadoop called LATE. The main difference between two schedulers is that LATE estimates the time left for a task to finish and uses this information for launching back-up task instead of using the progress of task as in original scheduler. They also carried out experiments to compare the performance between two schedulers. The result showed that LATE provided better performance in most of cases.

In another point of view, Chu and colleges [13] pointed out a characteristic that helps to recognize a class of machine learning algorithms which can be easily implemented with MapReduce. In more detail, algorithms that can write in summary form can easily adapt to run in multi-core environment. The authors also proposed a multi-core mapreduce framework which is based on the original architecture from Google. To prove their model is efficient, they conducted experiments to compare the performance of algorithms in two versions: one with mapreduce and the other in normal implementation. The result showed that the performance speed up very much, in some cases, it is linear with the number of the processing cores.

In heterogeneous or shared environments, data locality in scheduling of Hadoop may not be as good as desired. Therefore, the data transmission may occur in some tasks and causes performance degradation. To address this problem, Tao Gu et. al. proposed a data prefetching mechanism [31]. The idea is to minimize the overhead of data transmission. In Hadoop, the input data of map tasks is not transfered at one time, but in many small parts. When mapper get the first part of input, it processes that part of data. After finishing, it starts to transfers next part and processes it. This process is repeated till complete input is processed. This technique obviously wastes the time because data processing and transmission can be carried out in parallel. In data prefetching mechanism, a data fetching thread is created for requesting non-local input data and a prefetching buffer is allocated at the node which contains the task to store temporary data. Data prefetching thread retrieves input data through network and store in prefetching buffer. Map task only needs to process data in that buffer. Experiments result shows that this method can reduce data transmission time up to 95%, and improve 15% performance of the job.

Heterogeneous environment can also result in another problem with balanced data processing load. In more detail, more powerful node can complete its tasks on local data before a lower powerful node. Then it starts to handle tasks with data in remote slower node. Again data transmission becomes an overhead of system. Jiong Xie et. al. proposed a different approach [33]. They developed a data placement method in HDFS with two algorithms. The first algorithm deals with initial data placement: distribute data in a way

so that all the node can complete processing local data at the same time. It means high-performance node will contains more data. The second algorithm addresses dynamic data load-balancing problem when new data or nodes are added or data blocks are deleted. This algorithm re-organizes the input data. Experiments show improvement of their methods in two programs: Grep and WordCount.

In another approach to improve MapReduce performance, Zhenhua Guo and Geoffrey Fox tried to increase resource utilization [18]. In Hadoop cluster, each slave node hosts a number of task slots where tasks can run. When all slots are not fully used, the resources on idle slots are wasted. Authors introduced *resourse stealing* method, in which running tasks will steal idle resources by creating sub-tasks to share the workload. When master node assigns new tasks to a node, stolen resources on that node are returned back. The authors also introduced Benefit Aware Speculative Execution (BASE) method to schedule speculative task instead of default strategy of Hadoop. BASE estimates the remaining time of a task and predicts execution time of speculative task based on historical information. The speculative task is only executed if it can finish earlier than current task.

Improving Hadoop performance by data placing is a topic which attracts lots of researchers. Seo at. al. in their paper proposed a plug-in component for Hadoop called HPMR [29]. In HPMR, a prefetching scheme is used. In this prefetching technique, map or reduce tasks are carried out at the beginning of the input split, at the same time, data is prefetched from the end of the input split. This bi-directional technique simplifies the implementation of prefetching data. However, with prefetching approach, the task still has to fetch data by itself before it reaches the point where data are prefetched. HPMR also introduces a so-called pre-shuffling scheme. The idea behind this technique is simple: the pre-shuffling module examines the input split for map task and predicts the reducer which will process the output key-value pairs. Input data will be assigned to a map task which is near the predicted reducer to reduce the network transmission. To evaluate HPMR, authors conducted experiments on Yahoo Grid which consists of 1670 nodes. According to the reported result, performance is improved up to 73% with HPMR.

In the same vein of data placement, Rajashekhar and Daanish proposed two models to estimate the machine's power [11]. The mathematical model is built based on hardware specification like CPU speed, internal memory size, cache size, network speed, etc; while the history-based model bases on historical information of previous tasks like total input bytes, total output bytes, start time, end time. However, their evaluation does not show much about improvement of their solution. It mainly focuses on comparison between two suggested models.

1.1.2 Evaluating Parallel Solutions

Another branch of research focuses on evaluating existing parallel solutions to understand the actual performance on practical circumstances. From that, we have bases to choose suitable solutions for our issues. In [17], Ding et. al. conducted experiments to compare Hadoop, Hadoop Streaming and MPI with 6 benchmarks. The results show that the performance of Hadoop Streaming are often worse than the native Hadoop. The authors also pointed out that the overhead on Pipe Operations is one of the factor which causes performance degradation.

In [27], Pavlo et. al. compared Hadoop and two parallel database management systems (DBMS), DBMS-X and Vertica. Their results show that the performance of these DBMSs was better than Hadoop. In return, their processes to load data into took much longer

than Hadoop.

In [22], Kaur and his colleges presented a comparison of two open source frameworks for parallel computations, Hadoop and Nephele-PACT. The work focused on comparing the execution engine because both frameworks use HDFS. The results revealed that Nephele-PACT has better scalability while Hadoop shows its strength in fault-tolerance.

1.1.3 Extending Hadoop for non-Java Languages

Hadoop requires developers to write MapReduce programs in Java. Although Hadoop provides an utility to run applications written in other languages, called Hadoop Streaming, it suffers from a noteworthy degradation in the performance. As a result, several libraries are introduced to support writing programs without Java.

Abuin et. al. introduced Perldoop [10]. This is a tool that automatically translates Hadoop Streaming scripts written in Perl into equivalent Hadoop Java codes. Their performance comparison showed that Perldoop decreases the processing time with respect to Hadoop Streaming. Perldoop runs 12 times faster than Hadoop Streaming for Perl. However, some strengths of Perl language (for example, Perl is very good for pattern matching and regular expression) cannot be taken advantage of when translating original programs in Perl into Java codes.

Paper [23] represented a Python package Pydoop that provides a Python API for Hadoop MapReduce and HDFS. The package is built based on Hadoop Pipes (C++). Pydoop supports access to HDFS as well as the control of different components of MapReduce model such as combiner, partitioner, job counter, record reader and writer. The experimental results showed that Pydoop performs better than Hadoop Streaming but worse than Hadoop Pipes and the native Hadoop.

RHadoop [7,28] is an open source project developed by Revolution Analytics[1]. RHadoop contains four R packages which allow users to define map and reduce R functions. There are two important packages. The rmr package offers Hadoop MapReduce functionalities in R. The rhdfs package provides functions providing file management of the HDFS from within R. RHadoop is also an extension of Hadoop Streaming. User-defined functions in R will be called from Hadoop Streaming.

1.2 Motivation

Standard ML is a functional programming language which is mainly used in teaching and research. Although it is not widely used, Standard ML has been enormously influential. Many popular programming languages have adopted concepts of SML such as garbage collection, dynamically-scoped exceptions. SML is used to develop Automatic Design of Algorithms through Evolution (ADATE) [1]. This is a system for automatic programming. ADATE can automatically generate non-trivial algorithms. However, ADATE requires very large number of operations to run combinatorial search in order to employ program transformations. Therefore, ADATE consumes lots of time from hours to days to generate the desired programs. Reducing the execution time is the key point to bring ADATE to a wider community. Therefore, there is an urgent demand for distributed and parallel processing for Standard ML. This not only supports ADATE but also supports other systems which use SML as developing language.

[1] http://www.revolutionanalytics.com/

The first attempt to satisfy that demand is to take advantage of Message Passing Interface (MPI). For years, MPI has become a standard for writing parallel programs in research area. However, it suffers from a big problem. The effort that developers need to spend to write MPI programs is extremely large. With the introduction of MapReduce and its open implementation Hadoop, people have a second approach: integrating Standard ML with Hadoop.

Lots of research in the field of parallel processing focus on improving Hadoop's performance. However, there are only several efforts to evaluate different parallel approaches as well as develop new libraries (modules) to support programing languages other than Java to write Hadoop MapReduce programs. The problem is even more serious with developers who use SML because it is almost forgotten. Therefore, this thesis makes an effort to address those problems for SML developers. At first, we will develop a Standard ML API (library) for Hadoop. It supports writing MapReduce programs in SML that can run on Hadoop cluster. One of requirements for this library is that it must inherit good features of Hadoop MapReduce. In second part of this thesis, we will conduct experiments to compare new library with existing large-scale solutions. The result will be summarized and generalized into advice (guide-lines) to help developers to find the best appropriate approach for their issues in more general situation. In other words, the thesis will resolve following research questions:

1. How can we provide a new library that allows writing MapReduce programs in Standard ML?
2. How is the performance of new library in comparison with existing large-scale approaches?
3. What aspects should developers consider to find out the best solution for their actual issues?

1.3 Methodology

Existing efforts to provide a library for writing MapReduce programs in languages other than Java mainly follow two approaches: extend Hadoop Streaming [10] and Hadoop Pipes [23]. In the first approach, the library only allows developers to customize three main components of Hadoop: mapper, reducer and combiner. On the other hand, the second approach allows the library to support more useful features of Hadoop as mentioned in section 1.1.3. Therefore, this thesis uses the second approach to develop a Standard ML library for Hadoop.

In the evaluation, the new library will be evaluated with existing solutions which includes MPI, native Hadoop, Hadoop Streaming. All of these solutions will be tested in different metrics with different problems and under different cluster configurations.

1.4 Report Outline

The remaining of this thesis is organized as follows. Chapter 2 overviews the MapReduce framework and gives a short comparison with other systems. This chapter also describes Hadoop, an implementation of MapReduce in details. In chapter 3, concepts in Message Passing Interface are explained and some sample programs are presented. The Standard

ML language is introduced in chapter 4. Then the design of Standard ML library (MLoop) is described. Its implementation is also represented in this chapter. The chapter concludes by listing several limitations of MLoop. Chapter 5 compares the new library with other large-scale systems and discusses the result. Several guide-lines are also given at the end of this chapter. The final chapter gives a conclusion and suggests future work.

Chapter 2

MapReduce and Its Open Implementation Hadoop

The explosion of data in the digital era have brought a lot of opportunities to us. There is no doubt that big data has become a significant force for innovation and growth. In order to take advantages of big data, people must face with several fundamental challenges: how to deal with the size of big data, how to analyze it and create new insights as a competitive advantage. Since introduced in 2004, MapReduce and its open implementation Hadoop have become the de facto standard for storing, processing and analyzing huge volume of data. This chapter provides basic understanding about the MapReduce as well as the components of Hadoop. Through this chapter, we will know how to use Hadoop to solve different problems.

2.1 MapReduce

In 2004, Google published a paper that introduced MapReduce to the world. It is not a library or a programming framework, but instead is just a programming model for processing problems with huge datasets using a large number of computers. The model is inspired by the map and reduce functions which are commonly used in functional programming although they are used in MapReduce framework with a different purpose than that of the original.

2.1.1 Ideas Behind MapReduce

A lot of research has focused on tackling large scale problems. The ideas behind MapReduce are not quite new. They have been introduced and discussed in computer science theories for years. The point is that there is a tremendous combination of these ideas in MapReduce that brings into play the power of these ideas. Following are ideas that MapReduce obeys [25].

Scale "out", not "up". For processing massive amounts of data, a large number of low-end computer system (scaling "out" approach) is prefered to a small number of high-end servers (scaling "up" approach). The main reason for this is because investing in a such powerful system with high-end servers is not cost effective. Although there are issues with network-based cluster like communication between nodes, operational costs, data center efficiency, scaling out remains more attractive than scaling up.

Fault-tolerance. Failures are common on large cluster of low-end machines. For example, disk failures, errors in RAM, connectivity loss often happen. MapReduce are built on such cluster. Therefore, it is designed to be robust enough to handle failures which are common in a cluster. That is, it needs to have the capability to operate well without impacting the quality of service when failures happen. When a node goes down, another node will take responsibility of that node to handle the load. And after broken node is repaired, it should be able to re-join the cluster easily without manual configuration.

Move processing to the data. Instead of having "processing nodes" and "storage nodes" linked together via high speed interconnect like in high-performance computing applications, MapReduce moves the processing around. The idea is that we try to run the code on the processor which is on the same computer with the block of data we need. In that way, MapReduce can take advantage of data locality, which reduces the bottleneck in the network a lot.

Hide system-level details. System-level details (locking data structure, data starvation, parallelize the process) are hidden from developers. MapReduce separates the what and the how: what computation are to be performed and how they actually run on a cluster of machines. The programmers only need to care about the first part: what, and are free with the latter. Therefore, they can focus on designing algorithms or applications.

Scalability. For an ideal algorithm, scalability is defined with at least two aspects. In terms of data, given twice the amount of data, the same algorithm should take no more than twice the amount of time to finish. In terms of resources, when the cluster doubles the size, the same algorithm should run in no more than half as long. MapReduce is designed to reach this property although it is impossible to get exactly this. It can be considered that MapReduce represents a small step toward the above ideal algorithms.

2.1.2 MapReduce Basics

Funtional Programming Root

MapReduce is inspired by the *map* and *reduce* high-order functions in Lisp, or *map* and *fold* in other functional languages, as illustrated in Listing 2.1. *Map* function applies a given function f to each element of a list and returns a list of results. On the other hand, *fold* function takes a list, a function g of two arguments and an initial value. Fold can be left-associative (**foldl**) or right-associative (**foldr**). Left fold returns initial value if the list is empty. Otherwise, it applies g to initial value and the first element of list, then applies recursively left fold with result of previous step as initial value and the rest of original list as input list.

Listing 2.1: Map and Fold high-order functions

```
map f [a; b; c] = [f a; f b; f c]

foldl g initial []            = initial
foldl g initial [x1, x2, ...] = foldl g (g intial x1) [x2, ...]

// Example
```

```
fun f x = x * x
fun g (x,y) = x + y
map f [1,2,3,4,5] => [1,4,9,16,25]
foldl g 0 [1,2,3] => 6
```

The map phase in MapReduce corresponds to the map operation in functional programming, and the reduce phase corresponds to reduce or fold operation in functional programming. Each phase has a data processing function which is defined by user. These functions are called mapper and reducer respectively.

Programming Model

Key-value pairs are the basic data structure in MapReduce. Keys and values may be primitives such as integers, floating point values, strings, and raw bytes, or they could be user-defined types. The basic forms of mapper and reducer are as follows.

$map : (k1, v1) \longrightarrow [(k2, v2)]$
$reduce : (k2, [v2]) \longrightarrow [(k3, v3)]$

The mapper takes input key-value pair and produces a set of intermediate key-value pairs. The MapReduce framework then groups all intermediate values associated with the same intermediate key and passes them to the reducer. Intermediate data come to each reducer in order, sorted by the key. The reducer accepts an intermediate key and a set of values for that key. It processes these values and generates output key-value pairs.

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The pseudo-code for MapReduce program is shown in Listing 2.2.

Listing 2.2: Word Count with MapReduce

```
map(document_name n, document d):
    for each word w in document d:
        Emit(word w, count 1);
reduce(word w, counts [c1,c2,...]):
    sum = 0
    for each count c in [c1,c2,...]:
        sum = sum + c
    Emit(word w, count sum);
```

The map function takes input key-value pair in the form of (document name, document content) pairs, parses token (word) and produces an intermediate key-pair value for each word: the word itself as key and an associated count of occurrences (just 1 in this case) as value. All values of the same key are grouped together and sent to the reducer. Therefore, we just need to sum up all counts associated with a word. That is what is done in reducer.

MapReduce Execution

As described in original paper [16], the execution of MapReduce framework consists of six steps. When the user program calls the MapReduce function, the following sequence of actions occurs:

1. Input files are split into M pieces, typically 16-64MB per piece (this size can be controlled by user via a parameter). The MapReduce library then launches many copies of user program on the cluster.

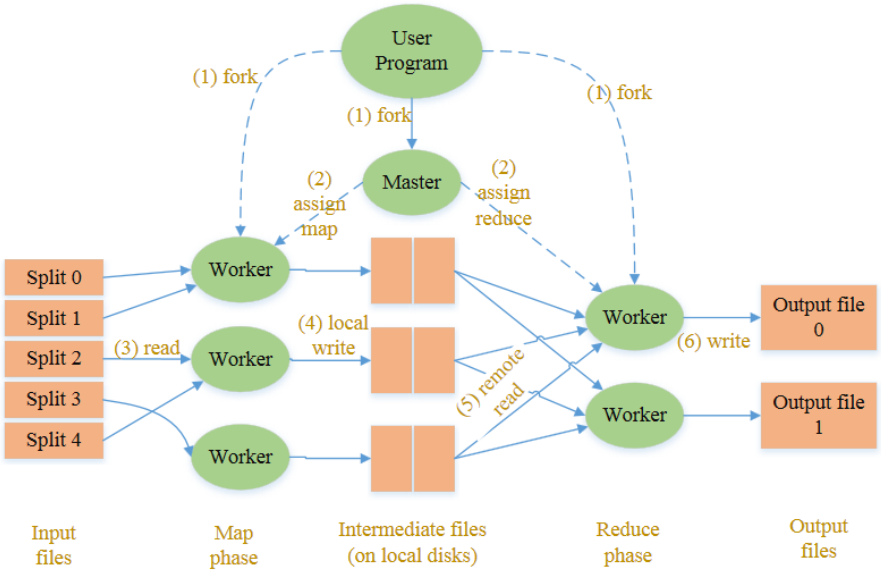

Figure 2.1: Execution Overview [16]

2. One of the copies of the program is the master, which assigns the work to the workers - the rest copies. There are total M map tasks corresponding to M pieces, and R reduce tasks (R is automatically determined by the MapReduce library or can be set by user). The master assigns a map task or a reduce task to each idle worker.

3. The worker who is assigned the map task reads the corresponding input split. It translates the input data into key-value pairs and passes each pair to the user-defined map function. The output key-value pairs are buffered in memory.

4. A background thread periodically spills the buffered data to local disk and splits them into R parts by the partitioning function, each part is corresponding to a reduce worker. The addresses of these data are reported to the master so that the reduce worker can know the location of these data through the master.

5. The reduce worker contacts the master to get information of its input data, then it uses remote procedure calls to read the intermediate data from the local disks of the map workers. After reading all necessary intermediate data, reducer sorts the data by the intermediate keys to group the data by the key. An external sort may be used if intermediate data cannot fit in main memory.

6. The reduce worker passes the key and the corresponding set of intermediate values to the user-defined reduce function. The output of this function is written directly to the output file system.

When the developer submits a MapReduce program (referred as a job) to the framework, it must take care of handling all aspects of distributed code execution on clusters. Responsibilities include [16, 25]:

Scheduling. Each MapReduce job is divided into smaller parts called tasks. Map task is responsible for processing a set of input key-value pairs, reduce task processes a set

of the intermediate data. When the total number of tasks is larger than the number of tasks that can be run in parallel on the cluster, the scheduler has to maintain some kind of task queue and track the state of running tasks. Another aspect involves multiple jobs from different users.

One more thing needs to care is something called *straggler*: a node takes unusual long time to complete one of the last few map or reduce tasks. This can lead to increasing total time for MapReduce operation. Therefore, the scheduler may need to schedule backup executions of the remaining in-progress tasks.

Data/Code Distribution. In MapReduce, the scheduler starts the task on the node which contains data for that task. That is "moving code to the data". If this is impossible (the node already had too many running tasks for example), new task is started on another node and the necessary data are streamed over the network.

Synchronization. In MapReduce, the reducer cannot start until all mappers have finished emitting key-value pairs and all intermediate key-value pairs have been shuffled and sorted.

Error and fault handling. MapReduce execution framework must ensure the smooth execution on a fragile environment where errors and faults are common. The master pings every worker periodically. If no response is received from a worker in a certain amount of time, the master marks it as failed. Any tasks completed and in progress on failed node are re-scheduled on other workers.

Partitioners and Combiners

Partitioners are responsible for splitting intermediate data and assigning them to reducers. The default partitioner uses hashing technique (compute the hash value of the key and then compute the modulo of this value with the number of reducers). In MapReduce, there is a guarantee that within a partition, the intermediate key-value pairs are processed in increasing key order. This makes it easy to generate a sorted output file per partition.

Combiners are an optimization of MapRedue that allows local aggregation before shuffle and sort phase. An example of this is the word count example. Since distribution of word occurrences are imbalanced, each map task may produce thousands of key-value pairs in the form <the, 1>. All of these intermediate data will be sent over the network to a reducer. This is obviously inefficient. One solution is to perform local aggregation on the output of each mapper.

Combiner function is executed on each machine that performs a map task. Combiners can be considered as "mini-reducers" that take the output of the mappers. The combiners can emit any number of key-value pairs but the keys and values must have the same type as the mapper output. Another difference between a reducer and a combiner is how its output is handled. The output of a reducer function is written to the final output file, whereas the output of a combiner is written to an intermediate file.

Figure 2.2 shows the complete MapReduce model. Input splits are consumed, translated to key-value pairs and processed by the mappers. Outputs of the mappers are handled by the combiners which perform local aggregations to reduce the number of intermediate output key-value pairs. The partitioner then splits the intermediate data, creates one partition for each reduce task. Based on this information, the reducer fetches its corresponding input data from mappers. When all the map outputs have been copied, the

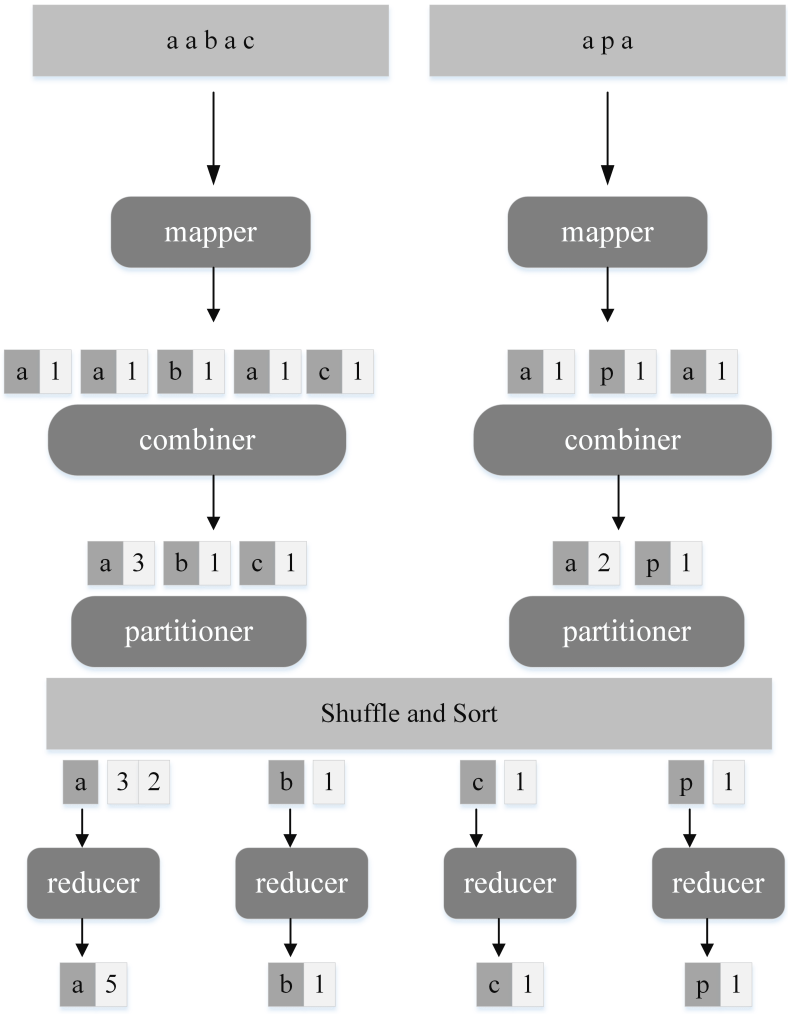

Figure 2.2: Complete view of MapReduce on Word Count problem

reducer starts the sort phase, which merges and sorts the map outputs. Finally, the reduce function is called to process the data and create the final output. In summary, a complete MapReduce job consists of the mapper, combiner, partitioner and the reducer, which are aspects that developers could manage. Other aspects are handled by the execution framework.

2.1.3 MapReduce Distributed File System

As described before, MapReduce is a programming model to store and process big mount of data. So far, we just focus on the processing aspect of the model. To deal with storing aspect, MapReduce uses a distributed file system (DFS), which builds on previous work and is customized for large-data processing workloads. The MapReduce DFS splits files into chunks, replicates them and stores them across the cluster. There are different implementations of DFS for MapReduce. The Google File System (GFS) supports Google's proprietary implementation of MapReduce; HDFS (Hadoop Distributed File System) is an open-source implementation of GFS that supports Hadoop; CloudStore is an open-source

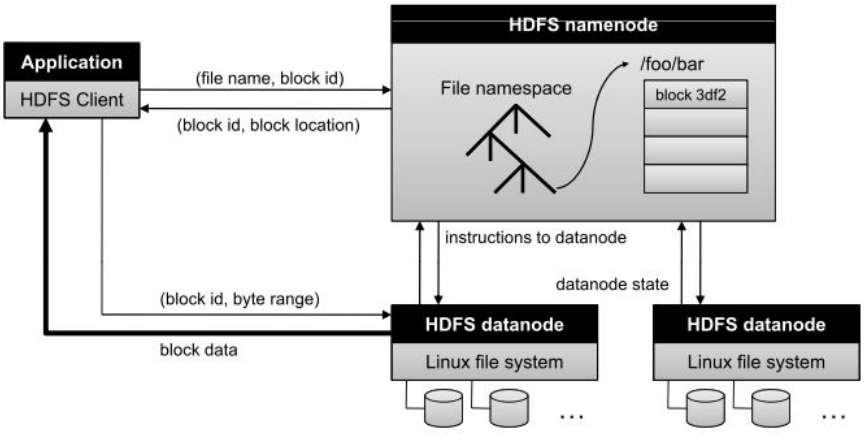

Figure 2.3: The architecture of HDFS [25]

DFS originally developed by Kosmix.

MapReduce DFS uses the master-slave architecture (Figure 2.3) in which master maintains a file namespace (meta-data, directory structure, file-block mapping, access permission, etc.) and the slaves store the actual data blocks. In GFS, the master is called the GFS master and the slaves are called GFS chunk-servers [25]. In Hadoop, they are namenode and datanodes respectively.

In HDFS, to read a file, the client must contact the namenode to get information about actual data. The namenode returns the block id and location of the block. Client then contacts datanode to retrieve actual data blocks. We should notice that all data are transfered directly between client and datanodes, data are never moved through the namenode. To ensure reliability, availability and performance, HDFS stores three copies of each data block by default. Datanodes are constantly reporting to the namenode to provide information about local changes as well as receive instructions to create, move or delete blocks from local disks. Namenode, therefore, can ensure proper replication of all the blocks: if there are not enough replicas, additional copies are created; if there are too many replicas, extra copies are removed.

2.1.4 MapReduce and Other Systems

The approach in MapReduce is not the only approach for parallel and distributed computation. There are other approaches which each of them is suitable for a specific class of problems. In this part, we will look at some of well-known approaches.

Relational Database Management System

Database Management System (DBMS) is a system that manages data, organizes it, and provides suitable ways to retrieve and process data. A relational database management system (RDBMS) is a DBMS that is based on the relational model.

We can use DBMS with a lot of disks to analyze large-scale data. However, this solution has several limitations. The first limitation comes from the disk drive technology: seek time is improved more slowly than transfer time. Seek time refers to the time required to position the disk's head on the correct track of the disk to read or write data, while

the transfer time mentions the time needed to transfer the data. If an operation on data needs more seek time than transfer time, reading or write a large amount of data will take longer than just streaming the data. In this case, MapReduce will be more efficient than B-Tree [30] (the data structure is common used in relational databases) because MapReduce uses Sort-Merge to rebuild the database. On the other hand, B-Tree is better for updating small portion of dataset. From this analysis, MapReduce is suitable for problems that need to work with whole dataset whereas RDBMS is good for problems which need low-latency or need to update a small fraction of data.

Another limitation of RDMBS is the type of data which it can manage. RDBMS works well with structured data, which are the data that conform a defined format. On the other hand, semi-structure data does not have a predefined schema and has a structure that may change unpredictably, while unstructured data are not organized in a predefine manner (plain text or image data are examples). RDBMS shows drawbacks when coping with semi-structured data or unstructured data. Fortunately, MapReduce works well on these kinds of data because it has the ability to interpret the data when processing them.

High Performance Computing and Grid Computing

Both of these systems aim at solving complex computation problems through parallel processing. High Performance Computing (HPC) relates to techniques to build and manage a supercomputer or parallel processing machine. There are two types of HPC architectures: Shared-Memory system and Distributed-Memory system. In shared-memory system, multiple CPUs will share a global memory. This architecture lacks of scalability between memory and CPUs. Adding more CPUs or transmitting a large amount of data will cause bottle-neck in data traffic. On the other hand, distributed-memory system has multiple nodes each with its own local memory. This architecture requires programmer to be responsible for mapping data structures across nodes. Programmer also has to manage communication between nodes when remote data is required in a local computation (message-passing). To perform message-passing between nodes, a low-level programming API (such as MPI) is needed.

Grid computing can be considered as a more economical way of achieving the processing capabilities of HPC. It connects computers and resources to form a cluster with higher performance. A widely used approach for communication between cluster nodes is MPI.

MPI - Message Passing Interface - is a language-independent protocol to program parallel computers. MPI allows programmers a lot of control. However, it is a low-level programming API. Therefore, it is hard to develop programs and requires programmers to handle lots of aspects: checkpoint, recovery, data flow, ...

Another difficulty which programmer may have to counter is to coordinate the processes in distributed environment. One of hardest aspect is to handle failures, especially partial failures - when you don't know exactly where the problem happens.

Volunteer Computing

Volunteer computing is a type of computing in which people (volunteers) provide computing resources to one or more projects. SETI@home [8] (analyze radio signal to find out the sign of life outside the earth) and Folding@home [6](analyze protein-folding and its effects) are two among well-known projects.

The volunteer computing projects work by breaking the problem into independent chunks called *work units*, which are sent to volunteers around the world to be computed [32]. Because volunteers are anonymous, volunteer computing systems must cope with problems related to correctness of the result. Besides, volunteers computing system requires time to solve and get the results from volunteers because they donate CPU resources, not network bandwidth.

Memcached

Memcached[1] is a free and open source, high-performance, distributed memory object caching system [14]. It is used to speed up dynamic database-driven websites by caching data in RAM. Many famous sites have used Memcached to increase their performance. Facebook, Twitter and Youtube are examples.

Indeed, Memcached is very simple. It uses a client-server architecture. The server maintains a key-value store and the client requests value from the server. Key and value are allowed to be any types. The server keeps these key-value pairs in RAM. If the RAM is full, it removes old pairs according to the LRU (Least recently Used) order.

One can use this technique to improve the performance of machine learning algorithms. It is possible. But for very large amount of data (or big data) and for long term use, Memcached is not a good choice because of some reasons. At first, Memcached is designed for caching system, it is not a cluster solution. Memcached nodes are independent and unaware of other nodes. Secondly, it lacks Scale-Out Flexibility [14]. Adding or removing a new node to/from an existing Memcached tier is a very complicated task. The next problem is the nature of RAM. When the system is offline, it forgets all the data. And finally, the cost of the RAM is much higher than hard disk. So it is not cost-effective to invest a Memcached system with large memory.

Apache Spark

Apache Spark[2] is originally developed in UC Berkeley. It is a open-source engine for processing large-scale data. Spark at first was written in Scala, then it was ported to Java and Python. Current version of Spark can easily integrate with Hadoop 2 cluster without any installation.

According to the Spark homepage, it is very efficient for iterative machine learning computations. Indeed, Spark outperforms Hadoop in this section. However, when the size of data increases and exceed the total size of RAM in the cluster by an order of magnitude, Hadoop, in turn, outperforms the Spark.

2.1.5 MapReduce Limitations

Although MapReduce provides lots of convenience to develop parallel programs. However, there are several limitations of MapReduce paradigm that needs to consider before applying it for practical issues.

Skew. In lots of cases, the distribution of intermediate key-value pairs is not balanced. As a result, some reducers must work more compared with others. These reducers become stragglers which make the MapReduce job slow down significantly.

[1] http://en.wikipedia.org/wiki/Memcached
[2] https://spark.apache.org/

Two phases. Some problems cannot be broken down into two-phases, such as complex SQL-like queries.

Iterative and Recursive Algorithm. Hadoop is built on cheap hardware. Therefore, failures are common. To deal with this problem, MapReduce restricts the units of computation: Both Map tasks and Reduce tasks have the *blocking property*: "A task does not deliver output to any other task until it has completely finished its work." Unfortunately, recursive tasks cannot satisfy this property. It needs to deliver some output before finishing because output of recursive task needs to feed back to its input. Therefore, to implement recursive task with MapReduce, we need to translate it to iterative form.

2.2 Hadoop: An Open Implementation of MapReduce

In 2002, Nutch project was started as a web search engine by Doug Cutting and Mike Cafarela. However, the architecture of Nutch at that time did not scale to the billions of web pages. MapReduce which was introduced in 2004 became the solution. In 2005, the Nutch developers finished a MapReduce implementation and ported the Nutch algorithms into MapReduce. In 2006, the storage and processing parts of Nutch were spun out to form Hadoop as an Apache project. In short, Apache Hadoop is an open source implementation of MapReduce written in Java. It provides both distributed storage and computational capabilities. In this section, we'll look at Hadoop in two aspects: its architecture and how to write programs in Hadoop.

Hadoop uses the master-slave architecture for both processing and storing model. At the time of this writing, Hadoop has two versions. The first version has several limitations in the scalability. The second version introduces new features that bring improvements for scalability and reliability of Hadoop. In the scope of this thesis, only Hadoop 2 is considered. It consists of following components:

- Yet Another Resource Negotiator (YARN), a general scheduler and resource manager. YARN enables its applications to run in Hadoop cluster, not just MapReduce.
- Hadoop Distributed File System for storing data.
- MapReduce, a computational engine for parallel and distributed applications. In Hadoop 2, MapReduce is a YARN application.

Hadoop provides a practical solution for large-scale data applications. A lot of projects are built on top of Hadoop to customize or optimize it for specific requirements. All of them form an ecosystem which is diverse and grows by day. Figure 2.4 shows the Hadoop ecosystem. Following is the brief description of several well-known projects:

Pig

A high-level data flow language and execution environment for exploring very large datasets. Pig runs on HDFS and MapReduce clusters.

Hive

A SQL-like data warehouse infrastructure. It allows analysts with strong SQL skills to run queries on the huge volume of data without any programming code in Java.

HBase

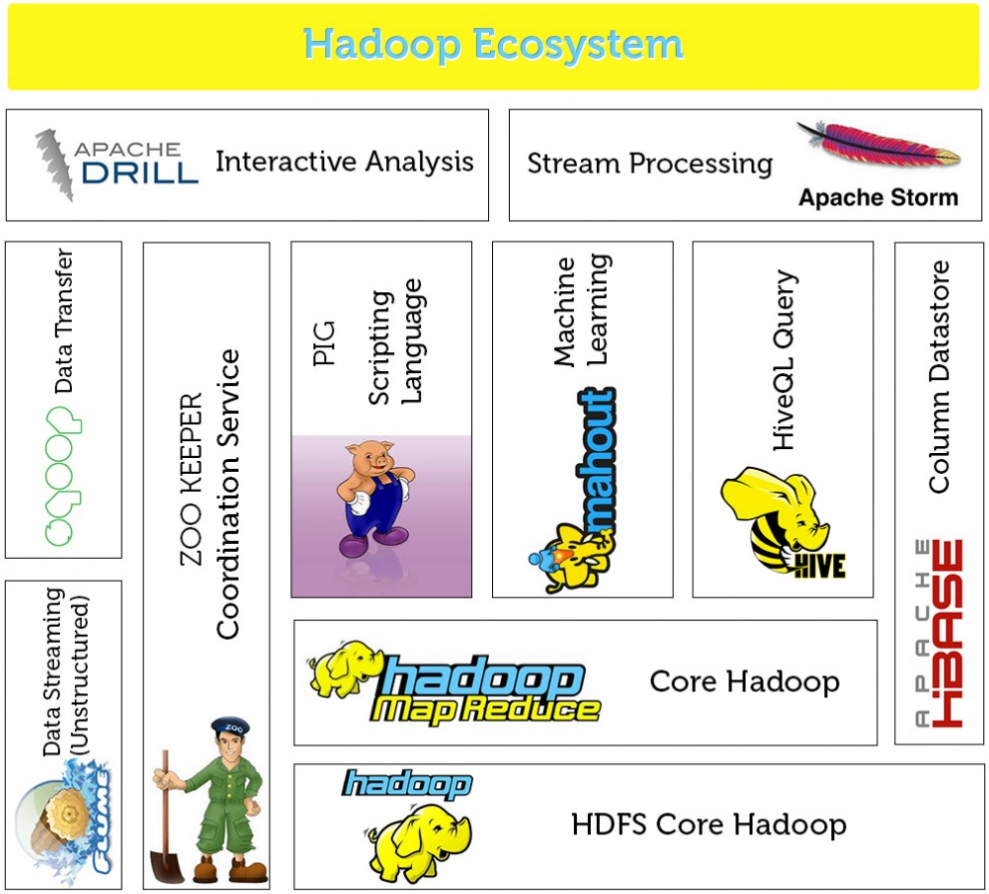

Figure 2.4: Hadoop Ecosystem

A distributed, column-oriented database modeled after Google's Bigtable. HBase is built on top of HDFS. HBase is suitable for applications that require real-time read/write random access to very large datasets.

ZooKeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. ZooKeeper steps up performance and availability of a distributed system such as a Hadoop cluster and protects the system from coordination errors such as race conditions and deadlock.

Mahout

A scalable machine learning libraries implemented on top of Hadoop. Mahout focuses on three key areas of machine learning: collaborative filtering (recommender engines), clustering and classification.

Storm

A distributed real-time computation system for processing large volumes of high-velocity data. Storm running on YARN is powerful for scenarios requiring real-time analytics, machine learning and continuous monitoring of operations.

2.2.1 Hadoop YARN

In Hadoop MapReduce v1, one JobTracker and multiple TaskTrackers are entities which manage the life-cycle of the MapReduce job. This, however, becomes a weakness of Hadoop because it limits the scalability of overall system. To overcome this limit, Hadoop v2 introduces a new architecture called YARN (Yet Another Resource Negotiator). YARN breaks main functionalities of the JobTracker into two independent daemons: a global ResourceManager to manage the resource in the cluster and many ApplicationMasters (one for each application) to manage the life-cycle of applications running in the cluster.

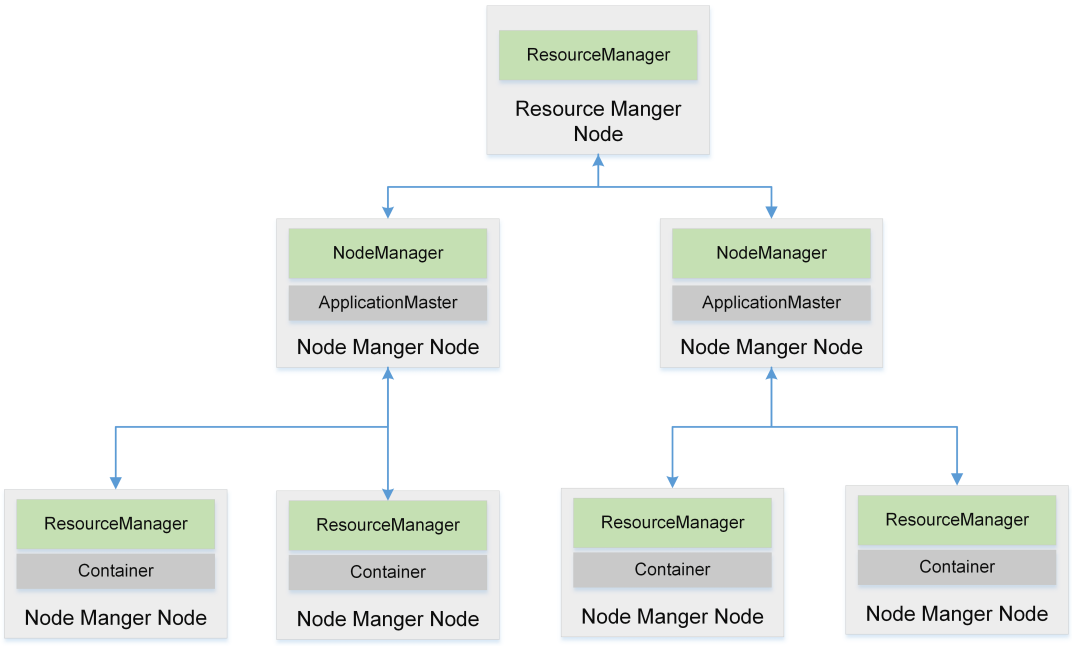

Figure 2.5: Hadoop YARN

The YARN framework has one primary duty, which is to schedule resources in a cluster. An application which wants to run in the cluster is managed by YARN and allocated containers (resources) which are specific for that application. When application is running, containers are monitored by YARN to ensure that resources are used efficiently. The YARN framework has two primary components:

- *Resource Manager.* A Hadoop cluster has only a Resource Manager. It is the master process of YARN which runs on master node of the cluster. Its only duty is to manage the resources of the cluster. It serves the client request for resources (which is defined as containers). If the sources are available, Resource Manager schedules the containers on the cluster and delegates the Node Manger to create the containers.

- *Node Manager.* The Node Manager is the slave process of YARN that runs on every slave node in the cluster. It is responsible for creating, monitoring and killing the containers. It serves requests from Resource Manager and Application Master container to create containers. The Node Manager also reports the status of containers to Resource Manager.

The architecture of Hadoop YARN is described in Figure 2.5. The central agent is ResourceManager which manages and allocates cluster resources. NodeManager runs on each node in the cluster; manages and monitors the resource allocation on the node. The ApplicationMaster runs under the schedule of ResourceManager and is managed by NodeManager. It is responsible for negotiating resources (which is defined as containers) from ResourceManager, also monitoring the execution and resource consumption of containers.

Running a Job in YARN

Running a MapReduce job in YARN involves many entities than classic MapReduce. They consist of the client, who submits the job; the Resource Manager, which schedules the resources on the cluster; the Node Manager, which launches and monitors the containers in cluster, the MapReduce Application Master, which manages the job and is responsible for fault-tolerance of the MapReduce application; the HDFS, which is the job resources.

The client sends request to ResourceManager. It then negotiates and allocates resources for a container and launches an ApplicationMaster there. Application Master is responsible for managing the submitted job which means managing the application-specific containers. The application master then keeps track of job's progress. It may request containers for all the map and reduce tasks in the job from the Resource Manager. Each map or reduce task is handled in a container. All requests contain information about the resources that each container requires such as which host should launch the container, the memory and CPU requirement of container. When task is running in container under YARN, it reports the progress and status back to its application master so that the application master has a overall view about the job.

The Application Master is also responsible for the fault-torrence of the application. The failure of container is monitored by Node Manager and reported to Resource Manager. When a container fails, the Resource Manager sends status message to the Application Master so that it can response to that event.

On job completion, the application master and the task containers clean up their working state and the resources are released.

Failures in YARN

There are many entities that can fail in YARN: the task, the application master, the node manager and the resource manager.

The task status is monitored by the application master. If task fails (task exits due to an running exception or task hangs), it is attempted to run again. The task is marked as failed after several attempts.

An application master sends periodic heartbeats to the resource manager. If the application master fails, the resource manager will detect the failure and start a new instance of the master running in a new container.

If a node manager fails, the resource manager will not receive heartbeats any more. Therefore, the resource manager will detect that failure. The failed node manager is removed from the pool of available nodes. Any task or application master running on the failed node manager is recovered as described above.

The failure of resource manager is serious. When running, the resource manager uses check-pointing mechanism to store its state to persistent devices. Therefore, the resource manager can be recovered from last saved state when a crash happens.

2.2.2 Hadoop Distributed File System

The Design of HDFS

HDFS is a filesystem designed for storing very large files which are maybe hundreds of gigabytes or terabytes in size. The idea for building HDFS is to provide the most efficient data processing. In detail, it is designed following write-one, read-many-times pattern. Moreover, Hadoop does not require expensive, highly reliable hardware. It can run on clusters of commodity hardware.

HDFS is not good fit for following requirements:

Low-latency data access. HDFS is designed to optimize data transfer. This may cause a high latency.

Lots of small files. The number of files in a filesystem which is managed by namenode is limited by the amount of its memory because namenode holds filesystem metadata in memory. Assume that each file, folder or block takes about 150 bytes. Then it needs at least 300 MB of memory to manage one million files.

Multiple writers, arbitrary file modifications. Files in HDFS are always written at the end of the file. Hadoop does not support for multiple writers or for modifications at arbitrary offsets in the file.

HDFS Concepts

Blocks

A disk block is a minimum amount of data in disk that can be read or written. Filesystem blocks are often several kilobytes in size while disk blocks are about 512 bytes. HDFS also use the concept *block*. However, its size is much larger: 64MB by default. The reason for this is to minimize the cost of seeking.

Namenode and Datanodes

HDFS adopts master-slave architecture with the *namenode* (the master) and lots of *datanodes* (the slaves). Namenode manage filesystem namespace while actual data are stored in datanodes. The client that want to access the data will communicate with the namenode to get information about actual data, then it contacts with datanodes to retrieve desired data blocks. Therefore, without the namenode, the filesystem cannot be used. As a result, it is very important to keep the namenode robust against failures. Hadoop provides two ways to ensure this.

The first thing is to back up the files. The second way is to provide a *secondary namenode*. This node differs from namenode in that it doesn't record any real-time changes to HDFS. Instead, it communicates with the namenode to take snapshots of HDFS metadata at each predefined interval. When the namenode is down, the secondary namenode will replace it and take the responsibility of namenode.

HDFS Federation

The namenode manages all filesystem namespace in memory. Therefore, in very large clusters, memory of namenode is a barrier of scalability. To deal with this, from Hadoop version 2.x, HDFS Federation [19] has been introduced, as shown in Figure 2.6. It allows

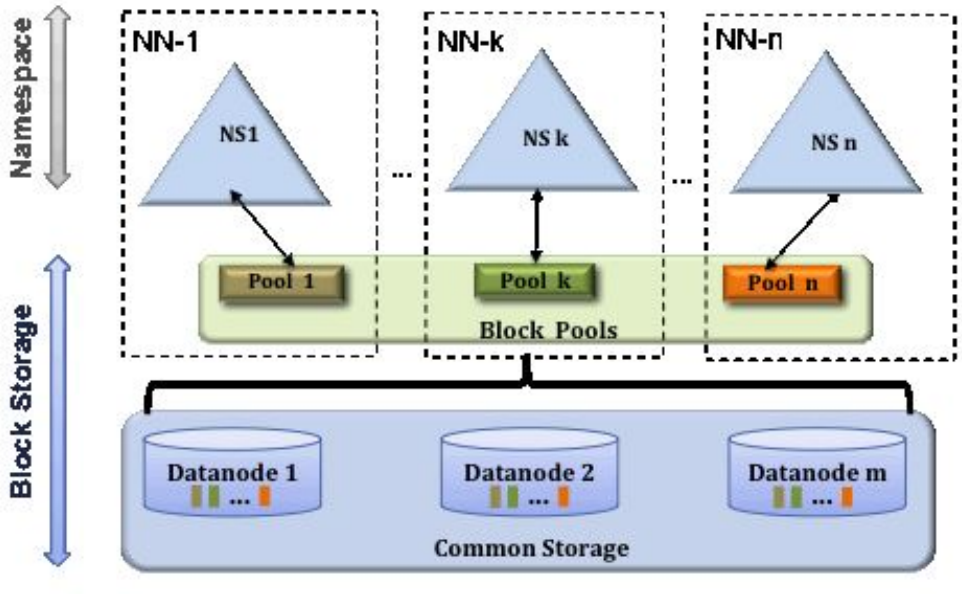

Figure 2.6: HDFS Federation

to expand the namenode by adding more namenodes, each of them will manage a proportion of file system. For example, one namenode manages all the files on folder /user, and another namenode controls files under /etc. The namenodes are federated, which means the namenodes are independent and don't require cooperation with one another. Therefore, the failure of one namenode does not affect other namenodes. The datanodes now are used as a commmon storage for all namenodes. Each datanode registers with each namenode in the cluster.

The block pool contains all the blocks for the files in a single namespace. It is managed independently with other block pools. So that a namespace can generate Block IDs for new blocks without cooperating with other namespaces.

HDFS High Availability

The namenode is the single point of failure of Hadoop. To deal with this problem, Hadoop 2.x introduces HDFS High-Availability, as shown in Figure 2.7. In this mechanism, two separated machines are used as namenodes in an active-standby configuration. It means that at any time, there is only one namenode which is in Active state, the other is in Standby. The role of the Standby namenode is to take over the duty of Active namenode quickly to continue responding to clients' requests. To do that, there are several requirements that the new Hadoop system must provide: Two namenodes have to share an edit log so that Standby namenode can update the current state of Active namenode; datanodes need to report their states to both namenodes because filesystem namespace is stored in main-memory, not the hard disk; when the failure occurs, the clients must have the ability to handle it without any intervention from users.

Hadoop introduces a new entity to manage the process of changing from Active namenode to Standby namenode. It is called *failover controller*. One responsibility is to ensure only one namenode is active at any time. The first implementation of failover controler uses ZooKeeper to do that. Failover process can be activated automatically by failover

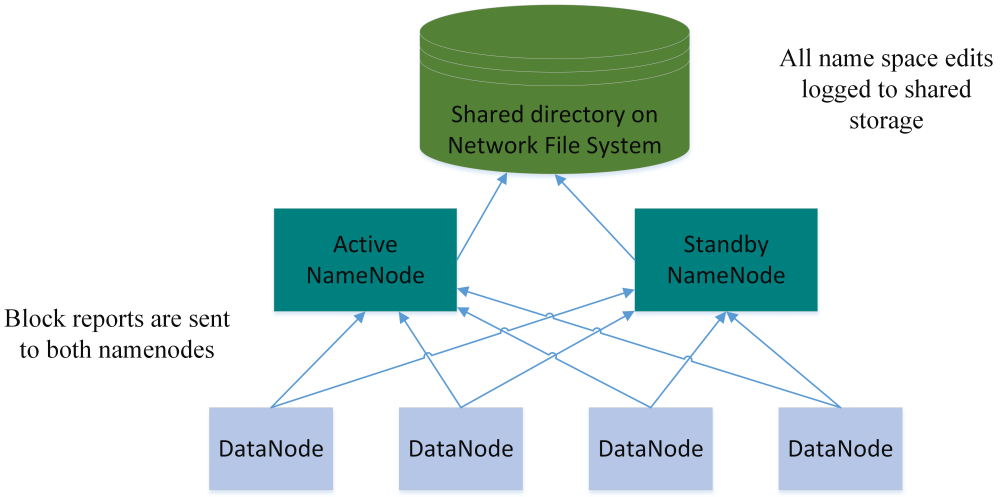

Figure 2.7: HDFS High Availability

controller or manually by administrator.

One notable point in failover process should be noticed is the likelihood of failed namenode has stopped working. Take a slow network as an example. This case can trigger a failover transition. However, the Active namenode is still running in reality. Therefore, to ensure that only namenode is in Active, a mechanism called *fencing* is applied. The failed namenode is forced to stop working by killing the namenode's process, revoking the privilege to access the shared storage directory.

2.3 Developing a MapReduce Application

In this section, we will take a closer look at writing MapReduce programs with Hadoop. A complete MapReduce program is often referred as a MapReduce job. A MapReduce job consists of mapper and reducer (also combiner and partitioner) as well as configuration parameters. Writing a MapReduce program mainly mentions functions for map and reduce step. The complete code can be found in Appendix B.

2.3.1 Counting words with Hadoop

The pseudo-code for the Word Count problem is listed in section 2.1.2. In this part, we will illustrate how to express it in Hadoop MapReduce. Our program mainly contains three parts: a map function, a reduce function and some code to configure and run the job.

Listing 2.3: MapReduce WordCount program

```
1 public class WordCount {
2     public static class WordCountMapper extends
3             Mapper<Object, Text, Text, IntWritable> {
4         private final IntWritable ONE = new IntWritable(1);
5         private Text word = new Text();
6
7         public void map(Object key, Text value, Context context)
8                 throws Exception {
```

```
9               StringTokenizer itr = new StringTokenizer(value.toString());
10              while (itr.hasMoreTokens()) {
11                  word.set(itr.nextToken());
12                  context.write(word, ONE);
13              }
14          }
15      }
16
17      public static class WordCountReducer extends
18              Reducer<Text, IntWritable, Text, IntWritable> {
19
20          public void reduce(Text text, Iterable<IntWritable> values,
21                             Context context) throws Exception {
22              int sum = 0;
23              for (IntWritable value : values) {
24                  sum += value.get();
25              }
26              context.write(text, new IntWritable(sum));
27          }
28      }
29
30      public static void main(String[] args) throws IOException,
31              InterruptedException, ClassNotFoundException {
32          Configuration conf = new Configuration(true);
33          Job job = Job.getInstance(conf, "wordcount");
34          job.setJarByClass(WordCount.class);
35
36          Path inputPath = new Path(args[0]);
37          Path outputDir = new Path(args[1]);
38          FileInputFormat.addInputPath(job, inputPath);
39          FileOutputFormat.setOutputPath(job, outputDir);
40
41          job.setMapperClass(WordCountMapper.class);
42          job.setReducerClass(WordCountReducer.class);
43
44          job.setOutputKeyClass(Text.class);
45          job.setOutputValueClass(IntWritable.class);
46          job.setInputFormatClass(TextInputFormat.class);
47          job.setOutputFormatClass(TextOutputFormat.class);
48
49          int code = job.waitForCompletion(true) ? 0 : 1;
50          System.exit(code);
51      }
52  }
```

Listing 2.3 shows a partial view of WordCount program. The map and reduce function are defined inside inner classes of WordCount. The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function. In this example, the input key is an Java object, the input value is a line (a text object), the output key is a word (a text object), and the output value is a number (an integer object). The `map()` method is passed a key and a value. The `Text value` contains the line of input. In line 9, we use Java StringTokenizer to tokenize the line based on whitespaces. The set of tokens then is looped over. Each token is extracted and cast into a Text object (line 11). To emit the key-value pair, we use an instance of `Context` which is provided by the `map()` method. In this case, we emit each cast token as *key* and a value of 1 as *value*. (line 12).

Four formal type parameters are also used to specify the input and output types of the reduce function. The input types must match the output types of the map function. Therefore, in this example, the input types of the reduce function are `Text` and `IntWritable`. The output type of the reduce function are defined as `Text` and `IntWritable`. The `reduce()` function receives the key (the word) and an— associated list of counts. It loops through the list and calculates the sum (line 22 - 25). Finally, it generates the final output key-value pair (line 26).

We now finish the definition of map and reduce functions. To make the job run, we also need to configure the job: what is the job's name, what are mapper and reducer of this job; where it should get the input files for the job and where it should store the output, ... The configuration is set up via a `Job` object. The job's name is "wordcount" (line 33). We pass a class to the `Job` in Job's `setJarByClass()` method. The job uses this information to locate the JAR file which containing our code.

The input and output paths of the job is specified by calling static methods `addInputPath()` on `FileInputFormat` and `setOutputPath()` on `FileOutputFormat` (line 36 - 39). We then specify the map and reduce class to use via `setMapperClass()` and `setReducerClass()`. The `setOutputKeyClass()` and `setOutputValueClass()` specify the output types for the map and reduce functions; which are often the same. In case of difference, we use `setMapOutputKeyClass()` and `setMapOutputValueClass()` to control the map output types.

The input and output types are controlled through `setInputFormatClass()` and `setOutputFormatClass()` methods. These classes specify how to parse to input files to key-value pairs for map function and how to write output key-value pairs from `reduce` function to output files.

Method `waitForCompletion()` return a Boolean value indicating success (true) or failure (false). We translate it into exit code (0 or 1).

2.3.2 Chaining MapReduce jobs

Above example illustrates a simple data processing task which a single MapReduce job can accomplish. However, there is always more complex tasks that a single job can not handle or have trouble in processing. These tasks may be split into simpler sub-tasks, which can be handled by single MapReduce jobs. For example, we may want to know top frequent words in a large document. A sequence of two MapReduce jobs can give us the answer. The first job calculate the number of occurrences of each word in the document. Then the second job filters out words with high number of occurrences.

Chaining jobs to run in sequence

We can chain MapReduce jobs to run in sequence, the output of one job becomes the input for the next job. There are several approaches to do that. However, the simplest approach is to use `JobConf` to configure the job and then pass the `JobConf` objects to `JobClient.runJob()` in order of the sequence. The `JobClient.runJob()` is blocked until it finishes the job. Therefore, the jobs will executed in the same order of the order of calling `JobClient.runJob()` methods.

Assume that we want to chain three jobs A, B, C to run sequentially. At first, we create `JobConf` objects for A, B, C. Then we configure all the parameters for each job. Finally, we submit them in sequence. The Listing shows the template of this.

Listing 2.4: Chaining job to run sequentially

```
// Create a JobConfs for job A, B, C, which are implemented by JobA, JobB,
    JobC classes respectively
JobConf jobA = new JobConf(new Configuration(), JobA.class);
JobConf jobB = new JobConf(new Configuration(), JobB.class);
JobConf jobC = new JobConf(new Configuration(), JobC.class);

// Configure parameters for JobA
jobA.setMapperClass...
jobA.setMapperClass...
...

// Configure parameters for JobB, JobC
...

// Submit jobs
JobClient.runJob(jobA);
JobClient.runJob(jobB);
JobClient.runJob(jobA);
```

Chaining jobs with complex dependency

When the jobs do not depend sequentially, we have to use a more complex approach. Hadoop provides a mechanism to do that via `Job` and `JobControl` classes. A Job object represents a MapReduce job. We already know how to initialize it through Word Count example. To specify the dependency between jobs, we use `addDependingJob()` method on `Job` object. For `Job` objects x and y, `x.addDependingJob(y)` indicates that job x only starts when job y finishes. Having all the dependencies of jobs, the execution of all the jobs is managed and monitored by `JobControl` object. Hence, we need to add jobs to `JobControl` object via `addJob()` method. Then `JobControl` object calls `run()` method to submit the jobs for execution.

Assume that we have a complex dependencies between jobs as shown in Figure 2.8. The arrow from node x to node y means that job y only runs after job x finishes. Listing 2.5 shows the template to specify these dependencies in Hadoop.

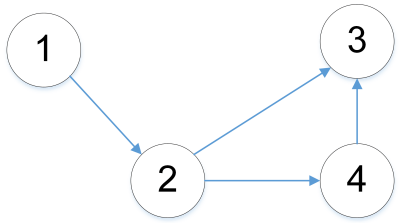

Figure 2.8: A complex dependencies between jobs

Listing 2.5: Chaining job with JobControl

```
// Create a JobControl
JobControl control = new JobControl("example");

// Initialize jobs
...
```

```
// Specifies the dependencies between jobs
job2.addDepending(job1);
job3.addDepending(job2);
job4.addDepending(job2);
job3.addDepending(job4);

// Add jobs to the JobControl object
control.addJob(job1);
control.addJob(job2);
control.addJob(job3);
control.addJob(job4);
...

// Submit jobs
control.run()
```

2.3.3 Example Applications with Hadoop MapReduce

This section introduces several programs written in Hadoop MapReduce. The full source-code can be found in Appendix section B.1.

Distributed Summation

The purpose of this application is to calculate formula: $1 + 2 + 3 + ... + 10^{10}$. The distributed calculation implemented with Hadoop will divide the task into ten sub-tasks, each for one map task. The result of summation is $50000000005000000000 = 5.10^9 + 5.10^{19}$.

The implementation in Hadoop is pretty simple. In the map function(), the value which is passed into contains two integers (start and end) in the text format separated by a blank. These values are passed into Big Integer objects (line 7-9). The mapper then calculates the sum from *start* to *end*, and emits an empty string as *key* and the sum as *value*. The reducer now gets the empty key and all the sum values associated with that key. It simply adds all these values to get the final result (line 21-24). The Java code is shown in Listing 2.6.

Listing 2.6: Mapper and Reducer for Summation example

```
1  class SumMap extends Mapper<LongWritable, Text, Text, BigIntegerWritable>{
2      Text word = new Text("");
3      protected void map(LongWritable key, Text value, Context context)
           throws IOException, InterruptedException {
4          String val = value.toString().trim();
5          if ("".equals(val))
6              return;
7          String[] temp = val.split(" ");
8          BigInteger start = new BigInteger(temp[0].trim());
9          BigInteger end = new BigInteger(temp[1].trim());
10         BigInteger sum = BigInteger.ZERO;
11         while(start.compareTo(end) <= 0){
12             sum = sum.add(start);
13             start = start.add(BigInteger.ONE);
14         }
15         context.write(word, new BigIntegerWritable(sum));
16     }
17 }
18
```

```
19 class SumReduce extends Reducer<Text, BigIntegerWritable, Text, Text>{
20     protected void reduce(Text key, Iterable<BigIntegerWritable> values,
           Context context) throws IOException, InterruptedException {
21         BigInteger sum = BigInteger.ZERO;
22         for (BigIntegerWritable value : values){
23             sum = sum.add(value.getValue());
24         }
25         context.write(new Text("Total sum: "), new Text(sum.toString()));
26     }
27 }
```

Graph Search

Graph theory is very famous today because of its application. There are many data structures to represent a graph. One way is to store using an adjacency list. Each node (vertex on the graph) stores a list of adjacent nodes.

BFS is a common strategy that is used for searching a graph. With this algorithm, lots of problem in graph theory are solved, for example: Find the shortest path in a graph, test a graph for bipartiteness. Given a graph represented using adjacency list. BFS colors each node white, gray or black. All nodes will start with white, means "not visited yet", then later become gray ("visiting") and then black ("visited"). Breadth-first search constructs a tree starting from a source node s, as followings. It colors s as gray, initials the distance d(s) = 0 and puts into an empty queue q. Then it repeats until q is empty: get one node n from q, get all white adjacent nodes of n, colors them as gray, sets distance to d(n) + 1, puts to the queue q and assigns n as black.

A single thread approach to implement BFS is good in many cases. But when working with a huge graph, it shows limitations. It's easy to find out that one problem is the lack of memory. Therefore, a solution to run this algorithm in parallel is essential.

An implementation of BFS with MapReduce can be found at the [21]. This site provides an interesting approach to implement BFS with map-reduce. In this implementation, a node is represented as:

```
ID    ADJACENT_NODES |  DISTANCE_FROM_SOURCE | COLOR
```

At beginning, all nodes have the distance is `Integer.MAX_VALUE`, source node has distance 0. An example of graph is:

```
1       2 | 0 | GRAY
2       1,3 | Integer.MAX_VALUE | WHITE
3       2 | Integer.MAX_VALUE | WHITE
```

Mapper will get all gray nodes. For each gray node, it emits new gray nodes with distance = current distance + 1 and ID is adjacent ID. Mapper also convert current gray node into black node and emits it. For all non-gray nodes, mapper emits them with no change (line 3-11, Listing 2.7).

```
1       2 | 0 | BLACK
2       NULL | 1 | GRAY
2       1,3 | Integer.MAX_VALUE | WHITE
3       2 | Integer.MAX_VALUE | WHITE
```

Because each time map function is called, it only receives a node, so that it doesn't know the adjacent nodes for the new gray node. So mapper leaves it as `NULL`.

Reducer will receive all data for a given key, which is all the copies of a given node. For example, for key = 2, corresponding reducer will gets:

```
2       NULL | 1 | GRAY
2       1,3 | Integer.MAX_VALUE | WHITE
```

Reducer will merge this information to create a final node: same ID, non-null adjacent nodes, minimum distance, darkest color (line 22-36, Listing 2.7). So, the output of first iteration is:

```
1       2 | 0 | BLACK
2       1,3 | 1 | GRAY
3       2 | Integer.MAX_VALUE | WHITE
```

With this approach, if we view the graph as the tree with start node as root node, one level of tree is processed after each iteration. After several iterations, all the nodes in the tree are traversed and the final output will be produced. This approach requires some way to store information of previous iteration because the program needs to know whether the last iteration achieves the final result. If it does, the program should stop. Fortunately, Hadoop allows developers to create their own custom *counters* to keep information of jobs. We take advantages of counter to store the number of gray nodes left after each iteration (line 43-44, Listing 2.7). In the main driver of the class, we check this counter and create a new job to process the last output if this value is greater than 0. Listing 2.8 shows the actual code in Java. The full program for this problem can be found in Appendix section.

Listing 2.7: Mapper and Reducer for Graph example

```
1 protected void map(LongWritable key, Text value, Context context) throws
      IOException, InterruptedException {
2     Node node = new Node(value.toString());
3     if (node.getColor() == Node.Color.GRAY) {
4         String edges = node.getEdges();
5         if (edges != null && !"NULL".equals(edges)){
6             for (String v : edges.split(",")) {
7                 context.write(new Text(v), new Text("NULL|" + (node.
                      getDistance() + 1) + "|GRAY"));
8             }
9         }
10        node.setColor(Node.Color.BLACK);
11    }
12
13    context.write(new Text(node.getId()), node.getLine());
14 }
15
16 protected void reduce(Text key, Iterable<Text> values,
17                               Context context) throws IOException,
                                      InterruptedException {
18    String edges = "NULL";
19    int distance = Integer.MAX_VALUE;
20    Node.Color color = Node.Color.WHITE;
21
22    for (Text value : values) {
23        Node u = new Node();
24        u.updateInfo(value.toString());
25        if (!"NULL".equals(u.getEdges())) {
26            edges = u.getEdges();
27        }
28
29        if (u.getDistance() < distance) {
30            distance = u.getDistance();
31        }
```

```
32
33          if (u.getColor().ordinal() > color.ordinal()) {
34              color = u.getColor();
35          }
36      }
37
38      Node n = new Node();
39      n.setDistance(distance);
40      n.setEdges(edges);
41      n.setColor(color);
42      context.write(key, new Text(n.getLine()));
43      if (color == Node.Color.GRAY)
44          context.getCounter(counter_enum.ITERATION).increment(1L);
45  }
```

Listing 2.8: Iterative jobs for Graph example

```
1  int iterationCount = 0;
2  long terminateValue = 1;
3  while (terminateValue > 0) {
4      String input;
5      if (iterationCount == 0)
6          input = "/input-graph";
7      else
8          input = "/output/output-graph-" + iterationCount;
9
10     String output = "/output/output-graph-" + (iterationCount + 1);
11
12     Job job = getJobConf(args);
13     FileInputFormat.setInputPaths(job, new Path(input));
14     FileOutputFormat.setOutputPath(job, new Path(output));
15
16     job.waitForCompletion(true);
17
18     Counters jobCnt = job.getCounters();
19     terminateValue = jobCnt.findCounter(counter_enum.ITERATION).getValue();
20     iterationCount++;
21 }
```

N-Queens

The N-Queens problem can be also solved with iterative Map-Reduce. In this problem, we combine the breadth-first-search and depth-first-search to solve.That is first five jobs find all possible ways of putting first five queens on board, then last job will find all solutions for each given board with first five queens placed. That is to use breadth-first search strategy (Job 1 - 5) followed by a depth-first search (Job 6). The code snippet can be found in Appendix section.

Listing 2.9: Mapper and Reducer for first five jobs of N-Queens

```
1  protected void map(LongWritable key, Text value, Context context) throws
       IOException, InterruptedException {
2      String val = value.toString().trim();
3      if ("START".equals(val)) {
4          for (int i = 0; i < n; i++) {
5              context.write(new IntWritable(1), new Text(String.valueOf(i)));
6          }
```

```
7         return;
8     }
9
10    String[] map = val.split("-");
11    int[] board = new int[map.length];
12    for (int i = 0; i < map.length; i++) {
13        board[i] = Integer.parseInt(map[i]);
14    }
15    Random r = new Random();
16
17    for (int row = 0; row < n; row++) {
18        if (safe(board, row)) {
19            context.write(new IntWritable(r.nextInt(10)), new Text(val + "-
                  " + row));
20        }
21    }
22 }
23 protected void reduce(IntWritable key, Iterable<Text> values, Context
       context) throws IOException, InterruptedException {
24    for (Text val : values) {
25        context.write(new Text(""), val);
26    }
27 }
```

Listing 2.9 shows the code for `map()` and `reduce()` method of first five jobs. The input of first job just contains a "START" string. The first job just generates all columns to put in the first row (line 3-8). The current board is stored as "1-3-5" means that in the first, second and the third columns, queens are put at the first, the third and the fifth rows respectively. Therefore, four next jobs need to passed this value to the board in map step (line 10-14). Then the mapper checks for all row to find all possible row to put in the next column (line 17-21). The reduce just emits all associated values it receives with an empty key.

Listing 2.10: Mapper for the last job of N-Queens

```
1 // parse the board
2 ...
3
4 Random r = new Random();
5 // 'len' is the number of queens appeared on board
6 int column = len; // column 'len' + 1
7 board[column] = -1;
8 while (column >= len) {
9     int row = -1;
10    do {
11        row = board[column];
12        board[column] = row = row + 1;
13    } while (row < n && !safe(board, column));
14    if (row < n) {
15        if (column < n - 1) {
16            board[++column] = -1;
17        } else { // found the board
18            String s = String.valueOf(board[0]);
19            for (int i = 1; i < n; i++) {
20                s += "-" + board[i];
21            }
22            context.write(new IntWritable(r.nextInt(10)), new Text(s));
23        }
```

```
24      } else {
25          column--;
26      }
27  }
```

The mapper for the last job is shown in Listing 2.10. From the current board, the mapper find all possible valid board to fulfill the board using depth-first-search (line 8-27). Each time it found a complete board, it emits the result (line 22). The last job uses the same reducer with previous jobs.

2.4 Hadoop Extensions for Non-Java Languages

Hadoop is built on Java. Hence, writing Hadoop MapReduce programs also requires Java to implement the algorithm. To support developers who do not want to use Java to write programs, Hadoop provides two extensions: Hadoop Streaming for all languages that support the standard input and output, Hadoop Pipes for C++.

Hadoop Streaming

Hadoop Streaming uses Unix pipes as the channel to communicate between Hadoop and user program. Therefore, it allows users to use any programming language that can read standard input and write to standard output to write MapReduce programs which can work with Hadoop cluster. Users provide map and reduce implementation through executables compiled by their languages. These executables read key-value pairs from standard input and write them to standard output. Data communicated between Hadoop framework and executables are based on Text Protocol: records are serialized as bytes of string. As a result, Hadoop Streaming is suitable for text processing. The Word Count program in python with Hadoop Streaming is shown in Listing 2.11 and 2.12.

Listing 2.11: mapper.py

```
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

Listing 2.12: reducer.py

```
#!/usr/bin/env python

from operator import itemgetter
import sys
```

```
current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    if current_word == word:
        current_count += count
    else:
        if current_word:
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Hadoop Pipes

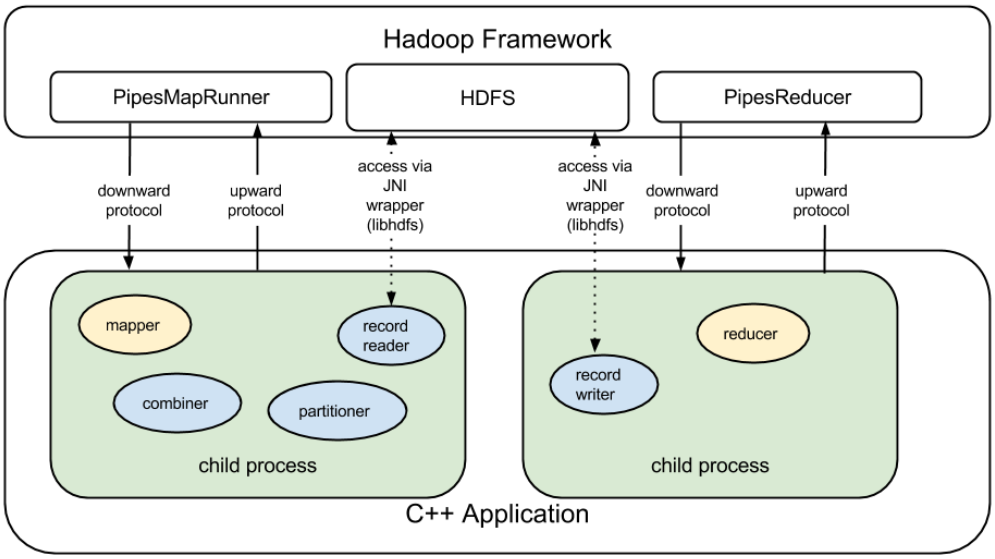

Figure 2.9: Hadoop Pipes data flows

Hadoop Pipes provides a C++ interface to Hadoop MapReduce. Hadoop Pipes uses sockets as the channel to communicate between Hadoop Framework and processes running C++ map or reduce functions. The data flows are shown in Figure 2.9. Hadoop uses Pipes Tasks to communicate with user-provided executables via persistent sockets.

C++ application provides a factory class used by the framework to create MapReduce components (Mapper, Reducer, RecordReader, Partitioner ...). When starting a Pipes Application, the framework sets up the PipeMapRunner and PipeReducer as well as locating the user-provided C++ executable file. When map phase starts, the framework automatically calls the PipeMapRunner and passes the input split into it. PipeMapRunner then creates a socket server to listen to C++ clients. When a socket is established, PipeMapRunner directs the client to process map task and processes the data returned from client via socket. The process is similar with PipeReducer for reduce tasks. With Hadoop Pipes, programmer can access and customize various MapReduce components in Hadoop. Furthermore, application is able to process any type of input data.

Chapter 3

Message Passing Interface - MPI

3.1 What is MPI

MPI is a library to make the work of programming parallel computers easier. MPI is an attempt to collect the best features of message-passing systems which have been developed over the years. There are several fundamentals about MPI which should be listed:

- MPI is a library, not a language. It specifies the names, calling sequences, and the results of routines called. The programs can be written in Fortran, C or C++ and are compiled by ordinary compiler and linked with the MPI library.
- MPI is a specification, not an implementation. Different parallel computer vendors offer their own MPI implementation for their machines. Well-known implementations are MPICH, OpenMPI, LAM/MPI.

3.2 Basic Concepts

This sections introduce basic concepts which need to be understood to begin with MPI. The points presented here are mainly gotten from the book *"An Introduction to Parallel Programming"* [26].

Rank is the common way to identify the process in parallel programming. It is a non-negative number. If there are p processes, then they will have ranks 0, 1, 2, ..., p-1, respectively.

3.2.1 MPI Program

Listing 3.1 shows a simple MPI program that uses *send* and *receive* messages.

Listing 3.1: A simple MPI send/recv program

```
1 #include <mpi.h> /* For MPI functions, etc */
2 #include <stdio.h>
3
4 int main(int argc, char** argv) {
5   MPI_Init(NULL, NULL);
6   int my_rank;      /* My process rank */
7   int comm_size;      /* Number of processes */
8   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
9   MPI_Comm_size(MPI_COMM_WORLD, &comm_size);
```

```
10
11   int number;
12   if (my_rank != 0) {
13     number = my_rank + 5;
14     MPI_Send(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
15   } else {
16     for (int i = 1; i < comm_size; i++){
17       MPI_Recv(&number, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
             );
18       printf("Process 0 received number %d from process %d.\n", number, i);
19     }
20   }
21   MPI_Finalize();
22   return 0;
23 }
```

If we compile this program and run it with two processes, the output would be

```
Process 0 received number 6 from process 1.
```

With five processes, the output would be

```
Process 0 received number 6 from process 1.
Process 0 received number 7 from process 2.
Process 0 received number 8 from process 3.
Process 0 received number 9 from process 4.
```

Let's take a closer look at the program. It starts with include declaration. To use MPI library in C program, it is required to include mpi.h header file. This contains macro definitions, type definitions and procedures of MPI. All of these start with the string `MPI_`.

3.2.2 MPI_Init and MPI_Finalize

`MPI_Init` tells the system to setup all the necessary stuffs, like allocate storage for message buffers (line 5). No other MPI functions should be called before `MPI_Init`. Its syntax is

```
int MPI_Init(int* argc_pointer, char*** argv_pointer);
```

The arguments `argc_pointer`, `argv_pointer` are pointers to the arguments of the `main()` function, `argc` and `argv`. Our program don't use these arguments, so we pass `NULL` for both.

`MPI_Finalize` tells the system that the program is done using MPI so that the system frees resources (line 21). No other MPI functions should be called after `MPI_Finalize`.

3.2.3 Communicators

A communicator in MPI is a collection of processes that can send messages to each other. The communicator called `MPI_COMM_WORLD` contains all of the processes started by the user. Information about the communicator can be retrieved through two functions:

```
int MPI_Comm_size(MPI_Comm communication, int* communication_size_pointer);
int MPI_Comm_rank(MPI_Comm communication, int* my_rank_pointer);
```

The first argument for both functions is a communicator, which has a MPI type, `MPI_Comm`. The number of processes in the communicator is returned to the second argument `communication_size_pointer` of the first function. The rank of calling

process in the communicator is returned to `my_rank_pointer` of the second function. Line 8 and 9 get information about `MPI_COMM_WORLD`, its size and the rank of calling process.

3.2.4 Communication

In the message-passing model, the processes which are executing in parallel have separate address spaces. Communication happens when a portion of one process's address space is copied to another process's address space. This occurs only when the one process executes a *send* operation and the other executes a *receive* operation. To send a message, `MPI_Send` is called. The syntax is

```
int MPI_Send(
  void*          msg_buf_p       /* in */,
  int            msg_size        /* in */,
  MPI_Datatype   msg_type        /* in */,
  int            dest            /* in */,
  int            tag             /* in */,
  MPI_Comm       communicator    /* in */);
```

The first three arguments define the content of message. The remaining arguments defines the destination of message. First argument is a pointer to the block of memory containing the contents of message. The second and third define the amount of data to be sent. The forth argument defines the rank of the process which receives this message. The fifth argument is a non-negative number. It is used to distinguish messages. Final argument is the communicator. It specifies the communication universe. A message sent by one process can be received by a process in the same communicator.

To receive the message, `MPI_Recv` is called. The syntax is:

```
int MPI_Recv(
  void*          msg_buf_p     /* out */,
  int            buf_size      /* in  */,
  MPI_Datatype   buf_type      /* in  */,
  int            source        /* in  */,
  int            tag           /* in  */,
  MPI_Comm       communicator  /* in  */,
  MPI_Status*    status_p      /* out */);
```

The first three arguments specify the memory available for receiving the message: `msg_buf_p` points to the block memory, `buf_size` is the available size of the block memory and `buf_type` indicates the type of object which stores the message. The next three argument identifies the message. The `source` specifies the the process which sends the wanted message. The `tag` and the `communicator` must match with the `tag` and `communicator` specified by the sending process. The last argument `status_p` contains information about the sending process, the tag of message and the amount of data in that message.

In our program, each process, other than process 0, creates a number (line 13) and sends it to process 0 (line 14). On the other hand, process 0 receives messages from other processes and prints out to the console (line 16-19).

3.2.5 Collective Communication

`MPI_Send` and `MPI_Receive` messages allow processes communicate with each other. These communications are often called **point-to-point** communications because we need

to specify the sending and receiving objects. There are situations that it is difficult to develop efficient programs with only above *send* and *receive* messages. For example, we want to calculate the global sum after each process has computed its partial value. The least efficient way to do is that each process with rank greater than 0 sends its value to process 0, then the process 0 calculates the sum of these values. We can instead do more efficient with a "tree structure" to calculate the sum as illustrated in Figure 3.1. In the first step, processes 1, 3 and 5 send their values to processes 0, 2 and 4, respectively. Then processes 0, 2, and 4 compute the sum of its value and received value. Subsequence steps is quite similar. Processes 2 and 6 send their values to processes 0 and 4, respectively. Processes 0 and 4 add received values to their values. Finally, process 4 send its value to process 0 to compute the final sum. This solution distributes the task of computing the sum to process 0, 2 and 4. Therefore, it reduces the task of process 0 from 6 additions to 3 additions. Moreover, it increases the degree of parallelization. For example, in the first step, additions in process 0, 2 and 4 can be conducted simultaneously.

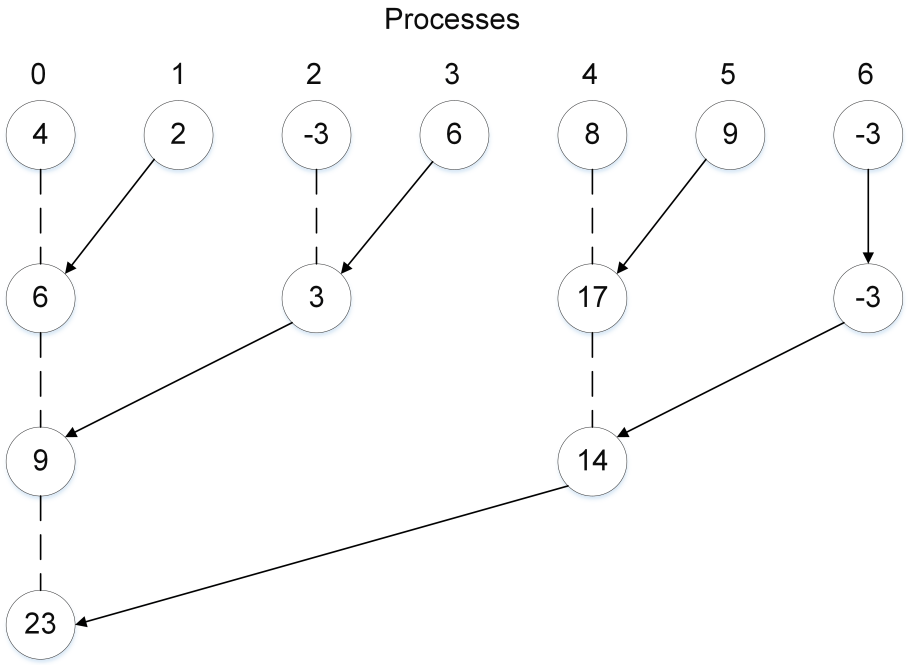

Figure 3.1: A tree-structured global sum

There are other possibilities to compute the sum in a "tree structure". It is hard to decide which option is best suitable for a specific case (for example, compute the sum of one hundred numbers, one million numbers). MPI, indeed, provides develop functions to simplify the tasks which require the communication of more than two processes. In MPI, a communication which involves all the processes in a communicator are called **collective communication**.

The global sum is just a case of collective communication. There are other cases, such as finding the maximum value, minimum value, or the product of values. All of these tasks can be achieved with a single function in MPI, `MPI_Reduce`.

```
int MPI_Reduce(
  void*         input_data_pointer    /* in  */,
```

```
  void*          output_data_pointer   /* out */,
  int            count                 /* in  */,
  MPI_Datatype   datatype              /* in  */,
  MPI_Op         operator              /* in  */,
  int            dest_proces           /* in  */,
  MPI_Comm       communicator          /* in  */);
```

The most important argument in this function is the fifth argument, `operator`. It defines the operation which is applied on input values to get the final output value. There are several pre-defined operators in MPI. Table 3.1 lists these operators. MPI allows developers to define their own operators. For the global sum , the operator is `MPI_SUM`. If each process keep its value in variable `double local` and we want to store the global sum into variable `double total`, the sum can be calculated by calling following function in each process.

```
MPI_Reduce(&local, &total, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Operator	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

Table 3.1: Predefined Operators in MPI

MPI also provides a variant of `MPI_Reduce` to support collective communication, but allows all the processes in the communicator to receive the final result. It is `MPI_Allreduce`.

```
int MPI_Allreduce(
  void*          input_data_pointer    /* in  */,
  void*          output_data_pointer   /* out */,
  int            count                 /* in  */,
  MPI_Datatype   datatype              /* in  */,
  MPI_Op         operator              /* in  */,
  MPI_Comm       communicator          /* in  */);
```

The arguments are identical except that there is no destination process required.

3.3 Sample Programs in MPI

To understand MPI, we write several programs with C++ using Open MPI library, an implementation of MPI, to solve different problems. These problems are the same with ones described in section 2.3. They are already implemented with Hadoop MapReduce. Now we'll see how to implement it in MPI. These MPI programs will be used in experiments reported in Chapter 5. The complete program can be found in Appendix section B.3.

3.3.1 Counting words

The first step to write this program is to define the parallel paradigm. In my implementation, one process reads the data and distributes it to other processes. Each process computes the word count on its part of data and reports the result to the master process. Here, the final output is computed. Then the next step is to implement the paradigm in actual code.

Listing 3.2: Read and distribute data

```
1  if (rank == 0) {
2      // assume number of line is equal to number of character / 10
3      pLineStartIndex = new long long int[buff_size / 10];
4      pLineStartIndex[0] = 0;
5      pszFileBuffer = readFile(file, total_size);
6  }
7
8  nTotalLines = 0;
9  if (rank == 0) {
10     // calculate the number of line in this chunk, store it in nTotalLines
11     // calculate the start index of each line in pszFileBuffer, store in
12     // pLineStartIndex array
13     // pLineStartIndex[1] stores the start index of the second line.
14 }
15
16 if (rank == 0) {
17     // get its part of data
18     ...
19
20     for (int i = 1; i < nTasks; i++) {
21         // calculate the data for each thread.
22         ...
23
24         // we need to send a thread the number of characters it will be
               receiving.
25         // curLength: the number of bytes of data.
26         MPI_Send(&curLength, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
27         if (curLength > 0)
28             MPI_Send(pszFileBuffer + pLineStartIndex[curStartNum],
                   curLength, MPI_CHAR, i, 2, MPI_COMM_WORLD);
29     }
30
31 } else {
32     MPI_Status status;
33     MPI_Recv(&totalChars, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
34     if (totalChars > 0) {
35         buffer = new char[totalChars + 1];
36         MPI_Recv(buffer, totalChars, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &
               status);
37         buffer[totalChars] = '\0';
38     }
39 }
```

Listing 3.2 shows the a part of code to read and distribute the input data. Process 0 (the master process) reads the file from hard disk, chunk by chunk (line 1-6). It then computes the appropriate parts of data for other processes and sends to them by calling two `MPI_Send` functions for each target process: one to specify the length of data, the other contains the actual data (line 17-29). Line 32-37 shows the code for receiving data

of processes with rank greater than 0.

Listing 3.3: Compute the word count on each part of data

```
map<string, int> wordcount;
int size = 0;
WordStruct* words = NULL;
if (buffer != NULL) {
    char* word = strtok(buffer, " ,.\r\n");
    while (word != NULL) {
        if (wordcount.find(word) == wordcount.end())
            wordcount[word] = 1;
        else
            wordcount[word]++;
        word = strtok(NULL, " ,.\r\n");
    }
    delete []buffer;
    size = wordcount.size();
    if (size > 0) {
        words = new WordStruct[size];
        int i = 0;
        for (map<string, int>::iterator it = wordcount.begin(); it !=
            wordcount.end(); it++) {
            strcpy(words[i].word, (it->first).c_str());
            words[i].count = it->second;
            i++;
        }
    }
}
```

After all the processes receive the data, the actual work is started: parse the word from the data, count the occurrence of each word as shown in Listing 3.3. The result then is sent back to the master process by calling `MPI_Send` functions. The final task for the master process is to aggregate the result and store it into the output file, Listing 3.4.

Listing 3.4: Aggreate the result

```
//aggreate the result from other processes
for (int i = 1; i < nTasks; i++) {
    int sz;
    MPI_Recv(&sz, 1, MPI_INT, i, 3, MPI_COMM_WORLD, &status);
    if (sz > 0) {
        WordStruct* local_words = new WordStruct[sz];
        MPI_Recv(local_words, sz, obj_type, i, 4, MPI_COMM_WORLD, &status);

        for (int j = 0; j < sz; j++) {
            totalwordcount[local_words[j].word] += local_words[j].count;
        }
        delete []local_words;
    }
}

// aggregate with local result
for (map<string, int>::iterator it = wordcount.begin(); it != wordcount.end
    (); it++) {
    totalwordcount[it->first] += it->second;
}
```

3.3.2 Summation

The first aspect that needs to be handled in this problem is that primitive types of C++ can not work with very big number. Therefore, we must use another library for this. In this program, GMP library (The GNU Multiple Precision Arithmetic Library[1]) is used. This is a free library which provides very good performance for arbitrary precision arithmetic.

The approach is that each process calculates the sum of a group of numbers. For example, assume that there are total 10 processes, then process 0 calculates the sum of numbers from 1 to $10^9 - 1$, process 1 calculates the sum of numbers from 10^9 to $2 * 10^9 - 1$ and so on. Each process then sends its result back to the process 0 to sum again and get the final result. Collective communication is not used because `MPI_SUM` does not work with special types of GMP.

Listing 3.5: Global sum with MPI

```
1 //initilize the parameter
2 mpz_init(sum);
3 mpz_t i;
4 mpz_init_set(i, start);
5 while (mpz_cmp(i, end) <= 0) {
6     mpz_add(sum, sum, i); // sum = sum + i
7     mpz_add_ui(i, i, 1);
8 }
9
10 if (rank == 0) {
11     // The master thread will need to receive all computations from all
           other threads.
12     MPI_Status status;
13     for (int i = 1; i < size; i++) {
14         int sz;
15         MPI_Recv(&sz, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
16         char* value = new char[sz];
17         MPI_Recv(value, sz + 1, MPI_CHAR, i, 2, MPI_COMM_WORLD, &status);
18         mpz_t pSum;
19         mpz_init_set_str(pSum, value, 10);
20
21         mpz_add(sum, sum, pSum); // sum += pSum
22     }
23 } else {
24     // The destination is thread 0.
25     char* t;
26     mpz_get_str(t, 10, sum);
27     int len = strlen(t);
28     MPI_Send(&len, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
29     MPI_Send(t, len + 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);
30 }
```

Listing 3.5 is a partial view of the global sum in MPI. Line 5-8 computes the sum of a group of numbers. The sum is then converted to string and sent to process 0 (line 25-29). Finally, process 0 gets the value, converts it back to GMP numbers and calculates the final sum (line 12-22).

[1] https://gmplib.org/

3.3.3 N-Queens

The N-Queens is solved in MPI with master-slave architecture. The master process will manage all the board the need to be solved. The worker (slave) processes actively request the board to solve from the master process whenever it is free (has nothing to do). The worker stops when there is nothing to solve any more.

Listing 3.6: The worker of N-Queen with MPI

```
1  while (true){
2      MPI_Send((void*) &REQUEST, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD);
3      MPI_Recv(&msg, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE
           );
4      if (msg == NO_MORE_WORK) break;
5
6      int newSize;
7      MPI_Recv(&size, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD,
           MPI_STATUSES_IGNORE);
8      if (size < 5)
9          newSize = size + 1;
10     else newSize = board_size;
11
12     board = new int[newSize];
13     if (size > 0)
14         MPI_Recv(board, size, MPI_INT, 0, DATA, MPI_COMM_WORLD,
               MPI_STATUSES_IGNORE);
15
16     //add a new queen, breadth first search
17     if (size < 5) {
18         for (int row = 0; row < board_size; row++) {
19             if (safe(size, board, row)) {
20                 board[size] = row;
21                 if (newSize == board_size) { // found complete solution
22                     //Store the board to local file
23                     ...
24                 } else {
25                     MPI_Send((void*) &NEW, 1, MPI_INT, 0, DATA,
                           MPI_COMM_WORLD);
26                     MPI_Send(&newSize, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD)
                           ;
27                     MPI_Send(board, newSize, MPI_INT, 0, DATA,
                           MPI_COMM_WORLD);
28                 }
29             }
30         }
31     } else {// depth first search: find the complete board
32         int column = size; // column 'size' + 1
33         board[column] = -1;
34         while (column >= size) {
35             int row = -1;
36             do {
37                 row = board[column] = board[column] + 1;
38             } while (row < board_size && !safeAtColumn(board, column));
39             if (row < board_size) {
40                 if (column < board_size - 1) {
41                     board[++column] = -1;
42                 } else { // found the board
43                     // store the board to local file
44                     ...
```

```
45                 }
46             } else {
47                 column--;
48             }
49         }
50 
51     }
52     delete []board;
53 }
```

Listing 3.6 shows the partial view of worker processes. It requests the board from the master process (line 2-14). If the board has less than five queens, it finds all possible solutions to put a queen in the next column (line 17-31). Each solution is reported to master process through message type NEW (line 25).

Listing 3.7: The master of N-Queen with MPI

```
1  while (listen) {
2      MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, DATA, MPI_COMM_WORLD, &
           status);
3      workerid = status.MPI_SOURCE;
4      if (msg == REQUEST) {
5          remaining = workQueue.size();
6          if (remaining > 0) {
7              iboard b = workQueue.front();
8              MPI_Send((void*) &NEW, 1, MPI_INT, workerid, DATA,
                   MPI_COMM_WORLD);
9              MPI_Send(&b.size, 1, MPI_INT, workerid, DATA, MPI_COMM_WORLD);
10             if (b.size > 0)
11                 MPI_Send(b.board, b.size, MPI_INT, workerid, DATA,
                       MPI_COMM_WORLD);
12             workQueue.pop();
13             delete []b.board;
14         } else {
15             // store free worker, then assign work for it later
16             freeWorker.push(workerid);
17             if (freeWorker.size() == nWorker) // all workers are free
18                 listen = false;
19         }
20     } else if (msg == NEW) {
21         MPI_Recv(&size, 1, MPI_INT, workerid, DATA, MPI_COMM_WORLD,
               MPI_STATUSES_IGNORE);
22         board = new int[size];
23         MPI_Recv(board, size, MPI_INT, workerid, DATA, MPI_COMM_WORLD,
               MPI_STATUSES_IGNORE);
24         if (size < board_size) {
25             // send this new work to free worker
26             if (freeWorker.size() > 0) {
27                 int id = freeWorker.front();
28                 MPI_Send((void*) &NEW, 1, MPI_INT, id, DATA, MPI_COMM_WORLD
                       );
29                 MPI_Send(&size, 1, MPI_INT, id, DATA, MPI_COMM_WORLD);
30                 MPI_Send(board, size, MPI_INT, id, DATA, MPI_COMM_WORLD);
31                 delete []board;
32                 freeWorker.pop();
33             } else { // store work and assign when requested
34                 iboard b;
35                 b.size = size;
36                 b.board = board;
```

```
37                  workQueue.push(b);
38              }
39          }
40      }
41  }
42  for (int i = 1; i <= nWorker; i++)
43      // stop signal to worker
44      MPI_Send((void *) &NO_MORE_WORK, 1, MPI_INT, i, DATA, MPI_COMM_WORLD);
```

The master process keep a queue, `workQueue`, to store all the boards that need to be solved, Listing 3.7. Each time a process contacts it to get new job, it pops a board and sends to that process (line 2-14). If there is no board left, it marks that process as free to assign work later as soon as it has new boards (15-18). If all other processes are marked as free, it knows that there is no change to get new boards. Hence, the process should stop (listen = false). It then sends the stop signal to other processes (line 42-44).

The new boards received from work processes (via `NEW` messages) are pushed back to the `workQueue` (line 34-37). Before doing so, the master process checks if there is free worker processes. If yes, the new board is sent immediately to a free process (line 26-33).

3.3.4 Graph Search

The approach to solve this problem is the same as described in previous chapter, section 2.3.3. In MPI implementation, process 0 (master process) reads the input file and distributes to other processes, each process has a part of file contains a subset of nodes in the graph. Each iteration is separated into four steps. In the first step, adjacency list for each node is parsed and new nodes are generated for each GRAY node (4-23). A hash function is used to determine which process this node should be sent to. Function `storeNewNode` adds the new node to list for that process. If the total nodes (the new nodes and nodes with color rather than GRAY) of the running process exceeds 1.5 million nodes, a small step is conducted to reduce the number of nodes with `reduce` function (line 25-33). This function simply does the job and aggregate values for all the node of the same ID to one node with minimal distance, darkest color and the known list of adjacent nodes. Listing 3.8 shows the code.

Listing 3.8: MPI - Graph Search (step 1): Generate new nodes from GRAY node

```
1  while (newLoop) {
2      newLoop = false;
3      // Step 1: map data
4      for (int i = 0; i < size - 1; i++)
5          other_data[i].clear();
6      int pack = 0;
7      for (int i = 0; i < input.size(); i++) {
8          vector<string> node = parseNode(input[i].c_str());
9          if (node[3].compare("GRAY") == 0) {
10             vector<string> edges = split(node[1].c_str(), ',');
11             for (int i = 0; i < edges.size(); i++) {
12                 string new_node;
13                 string newDist;
14                 increaseDist(newDist, node[2]);
15                 createNode(new_node, edges[i], "NULL", newDist, "GRAY");
16                 storeNewNode(size, edges[i], new_node, other_data_tmp);
17             }
18             string oldNode;
```

```
19              createNode(oldNode, node[0], node[1], node[2], "BLACK");
20              storeNewNode(size, node[0], oldNode, other_data_tmp);
21              pack += edges.size();
22          } else
23              storeNewNode(size, node[0], input[i], other_data_tmp);
24          pack++;
25          if (pack >= 1500000) {
26              pack = 0;
27              for (int i = 0; i < size; i++) {
28                  other_data_tmp[i].insert(other_data_tmp[i].end(),
                        other_data[i].begin(), other_data[i].end());
29                  other_data[i].clear();
30                  reduce(other_data_tmp[i], other_data[i]);
31                  other_data_tmp[i].clear();
32              }
33          }
34      }
35      ...
36  }
```

In the second step, data is re-distributed between processes, as shown in Listing 3.9. The purpose of this is to have all the nodes of the same ID are sent to the same process. We then sort the data by node id to group by node id so that the aggregation step is able to be done right.

Listing 3.9: MPI - Graph Search (step 2,3): Gathering data and sorting

```
1  while (newLoop) {
2      ...
3      // Step 2: send data to corresponding processes...
4      for (int i = 0; i < size; i++) {
5          reduce(other_data_tmp[i], other_data[i]);
6          other_data_tmp[i].clear();
7      }
8      // for each phase, 1 process sends, others receive.
9      for (int phase = 0; phase < size; phase++) {
10         if (phase == rank) {// send
11             for (int i = 0; i < size; i++) {
12                 if (i == rank)
13                     continue;
14                 sendData(i, other_data[i], other_data[i].size());
15                 other_data[i].clear();
16             }
17         } else {// receive
18             receive_resp(phase, other_data[rank]);
19         }
20     }
21
22     // Step 3: Sort data
23     sort(other_data[rank]);
24 }
```

In the last step, we aggregate all the nodes with the same ID to create a final node of this iteration (line 4-44). We then update the condition for the loop. We continue the loop if all worker processes do not generate any new GRAY nodes (line 45). Otherwise, a new iteration is started with the same manner. The partial view of this step is shown in Listing 3.10.

Listing 3.10: MPI - Graph Search (step 4): Aggregate the data

```
while (newLoop) {
    ...
    // Step 4: find the smallest distance
    input.clear();
    string previous("NULL");
    string edges("NULL");
    string distance("Integer.MAX_VALUE");
    string color("WHITE");
    bool hasGray = false;
    for (int k = 0; k < other_data[rank].size(); k++) {
        vector<string> node = parseNode(other_data[rank][k].c_str());
        string node_id = node[0];

        if (node_id.compare(previous) != 0) {
            if (k > 0) {
                string nnode;
                createNode(nnode, previous, edges, distance, color);
                input.push_back(nnode);

                if (color.compare("GRAY") == 0)
                    hasGray = true;
            }
            previous = node_id;
            edges = "NULL";
            distance = "Integer.MAX_VALUE";
            color = "WHITE";
        }

        if (node[1].compare("NULL") != 0) {
            edges = node[1];
        }
        minDist(distance, distance, node[2]);
        maxColor(color, color, node[3]);
    }
    if (other_data[rank].size() > 0) {
        string nnode;
        createNode(nnode, previous, edges, distance, color);
        input.push_back(nnode);
    }

    other_data[rank].clear();

    if (color.compare("GRAY") == 0)
        hasGray = true;
    MPI_Allreduce(&hasGray, &newLoop, 1, MPI_BYTE, MPI_BOR, MPI_COMM_WORLD)
        ;
}
```

Chapter 4

Standard ML API for Hadoop

Standard ML (SML) is a functional programming language with compile-time type checking. It supports polymorphic type inference which frees developer from specifying types of variables. Like all functional programming languages, the function is a main feature of Standard ML. Programming with recursive and symbolic data structure is common in SML with the pattern matching feature of function definitions. SML also provides techniques to structure large programs: module hierarchies, enforce abstraction or build generic interfaces.

There are different compilers for compiling a program written in SML. Standard ML of New Jersey (SML/NJ) is a full compiler with associated tools and libraries. It supports an interactive shell. MLton is a whole-program optimizing SML compiler that produces very fast executables compared to other SML implementations. The executables also have smaller sizes. Furthermore, MLton supports communication between C and SML via a special feature. Moscow ML is a light-weigh implementation which implement the full SML language.

This chapter introduces basic knowledge about SML. It also analyzes advantages and disadvantages of some current large-scale solutions for SML. The approach to provide a new library for SML for large-scale problems is then described. This library is named MLoop. Its architecture and implementation are reported in Section 4.4. Finally, Section illustrates how to solve several problems with MLoop.

4.1 Programming in Standard ML

4.1.1 Type, Values and Expression

A type in Standard SML is defined by specifying three things: the name, the possible values and possible operations on those values. Take the type of integer as an example. Its name is int. Possible values are whole numbers 0, 1, ~1, 2,~2 and so on (~ is used for minus sign in SML). And possible operations include + (addition), - (subtraction), * (multiplication), div (quotient) and mod (remainder).

A value is an atomic expression which cannot be evaluated any further. Compound expressions include atomic expression and expressions built by applying an operator on other compound expressions. Every expression has to have a type; then it is said to be well-typed. Otherwise, it is said to be ill-typed; which is considered in-eligible for evaluation. The evaluation of expressions is governed by a set of rules, called evaluation rules. These

rules define how to determine the value of a compound expression as a function of its constituent expressions. For example, 2 + 3 is evaluated as value 5 because applying operation + to two atomic expressions 2 and 3 yields 5.

4.1.2 Variables and Declarations

Like many programming languages, values can by assigned to variables. However, unlike other languages, variables in ML do not vary. A value is bound to a variable via a construct called value binding. This can be seen as an assignment in other languages. But the value is bound to the variable forever. So variables in SML are similar to constants in C or Java. Therefore, variables in ML are close to the definition of variables in mathematics than to variables in languages such as C or Java.

A type may be also bound to a type constructor via type binding. A binding generates a new variable that effects within its scope. For example: *type float = real* introduces a type variable float which is synonymous with real. Similarly, *val m:int = 3 + 3* introduces the variable m with type int and value 6. One important thing to remember is that bindings are not assignments. Once a variable is bound to a value, the binding of that variable never changes.

A binding is an atomic declaration. Multiple declarations can be written one after the others.

4.1.3 Mutable Data Structures

Coding with immutable variables provides comfortableness. Because we are free from side effects and therefore, it's easier to reason about the behavior of the code. However, in some situations, we need mutable data structures to do our purposes more effectively. For example, counting the number of words that already appeared. When a new word comes, we need to update the current count.

Like most imperative programming languages, SML supports mutable data structures. There are two built-in ones in SML: Reference and Array. Type `typ ref` is the type of reference cells containing values of type `typ`. A value of type `int ref` is a pointer to a location in memory, where that location contains an integer. It's similar to `int*` in C/C++. Like all values, reference cells may be bound to variables, passed as arguments to functions, returned as results of functions, etc...

A reference cell is created by the function `ref` of type `typ -> typ ref`. The content of a reference cell is retrieved using the function `!` of type `typ ref -> typ`. By using the `:=` operator, the value in the cell can be changed as a side effect. Here are examples.

```
val x : int ref = ref 3
val y : int = !x
val _ = x := (!x) + 1
val z = y + (!x)
```

The first line creates a reference cell with content of 3. The second line reads the content of the cell referenced by x, then binds it to y. So y has value 3. The third line evaluates "!x" to get 3, adds one to it to get 4, then updates the content of reference cell by x to this value. The final line gets the content of reference cell by x and adds to y. Therefore, it yields 3 + 4 = 7 and binds to z.

Array is another mutable data structure that SML provides. Arrays generalize the *reference* concept in that they are a sequence of memory locations. Basic operations of

arrays are provided via `Array` structure.

4.1.4 Type Inference

In most programming languages, we must explicitly assign it a type. ML allows us to omit this type information whenever the compiler can infer it from the context when declaring a variable. This process is called type inference. For example, the value binding *val v:int = 5* can just be written as *val v = 5* because the type of value 5 is int.

This feature enables functions in SML to have a characteristic called *polymorphic*. The identity function *fn x=>x* is an example. The body of the function has no constraints on the type of x. This expression is not rejected because it works perfectly for any choice of the type of x. This is contrast to the function *fn x =>x + 1* which there are only two possible types of x (int or real). The choice of int or real affects the behavior of the function: perform an integer addition or a floating point addition. The behavior of the identity function is uniform for any types of its argument, which is said to be polymorphic.

4.1.5 Higher-Order Functions

Functions which take functions as arguments or yields functions as results are known as higher-order function. The `map'` function below is an example of function which receives another function as argument.

```
fun map' (f, nil) = nil
| map' (f, h::t) = (f h) :: map' (f,t)
```

For example,

```
map' (fn x => x + 1, [1,2,3,4])
```

returns the list [2,3,4,5]

The `add3` function below is an example of function which returns a function.

```
fun add3 a = fn () => a + 3
```

has the type `int -> (unit -> int)`. The value of `add3 7` is the function which receives no arguments and yields `7 + 3`. So `add3 7 ()` yields `10`.

4.1.6 Signature and Structure

Signature and Structure are employed to construct module in ML. Signature and structure can be seen as interface and class respectively in languages such as C++ or Java.

A structure is a unit of program which consists of a sequence of declarations of types, exceptions, variables and functions. Below is an example of structure

```
structure IntLT = struct
type t = int
val lt = (op <)
val eq = (op =)
end
```

This structure has three components, one type and two functions. The component of structure can be accessed using paths. For example, `IntLT.lt` refers to function `lt` of structure `IntLT`: `IntLT.lt (3,5)` yields `true`.

Signature is the type (or specification) of a structure. A signature specifies some requirements on a structure which implements it, such as the type component, the value components that structure must have. Here is an example,

```
signature ORDERED = sig
type t
val lt : t * t -> bool
val eq : t * t -> bool
end
```

This signature specifies a structure that provides a type component named `t` and two functions with type `(t * t) -> bool`.

Signature may be built from another using signature inclusion and signature specialization. These forms are signature inheritance. Signature inclusion is used to add more components to an existing signature.

```
signature ORDERED_CLONE = sig
include ORDERED
end
```

This signature just includes signature `ORDERED` with no additional components.

Signature specialization is used to clarify an existing signature with additional type definitions

```
signature INT_ORDERED =
ORDERED where type t = int
end
```

4.1.7 Signature Ascription

Signature ascription expresses the requirement that a structure implements a signature. There are two forms of signature ascription, transparent and opaque.

```
structure IntDiv : ORDERED = struct
type t = int
fun lt (m, n) = (n mod m = 0)
val eq = (op =)
end
```

Above transparent ascription is so-called because the synonym between `IntLT.t` and int is still valid in the scope of this structure.

Here's an example of opaque ascription. We may use opaque ascription to specify that a structure implement queues, and, at the same time, specify that only the operations in the signature be used to manipulate values of that type.

```
signature QUEUE = sig
type 'a queue
val empty : 'a queue
val insert : 'a * 'a queue -> 'a queue
exception Empty
val remove : 'a queue -> 'a * 'a queue
end

structure Queue :> QUEUE = struct
type 'a queue = 'a list * 'a list
val empty = (nil, nil)
fun insert (x, (bs, fs)) = (x::bs, fs)
exception Empty
fun remove (nil, nil) = raise Empty
| remove (bs, f::fs) = (f, (bs, fs))
| remove (bs, nil) = remove (nil, rev bs)
end
```

In this structure, the definition of `'a Queue.queue` is hidden by the binding; the equivalence of the types `'a Queue.queue` and `'a list * 'a list` is not propagated into the scope of the binding.

4.2 Parallelization for SML

Standard ML is mainly used for teaching and research. However, it is not a widely used language. Therefore, there is not much effort to provide extensions/solutions for SML to adapt to new demands of changes in technologies. Parallel and distributed processing is an example. The explosion of large-scale applications which are not handled by a single server require a solution for Standard ML that can take advantage of parallel processing in cluster.

Currently, there are two well-known approach for parallelization for SML that can work with a cluster. The first choice is to use MPI. In this approach, we manage the parallel architecture with MPI C/C++. Then we can call the actual code in SML to do the task. However, this approach suffers from several issues. At first, MPI provides no fault-tolerance ability. When one process fails, the overall task also fails. The burden of handling the fault is given to developers. Secondly, MPI only provides the way to communicate between processes. Developers have to manage the parallel paradigm by themselves. This task is really difficult and costs a lot of time.

The second approach is to use Hadoop. As introduced in Chapter 2, Hadoop provides a lot of good features. It allows developers to focus on application algorithm while the system-level detail of parallel processing is hidden and managed by Hadoop. It also offers scalability and robustness to the system. However, Hadoop is written in Java and provides Java APIs to interact with both MapReduce and HDFS. Therefore, to use all of the features that Hadoop provides, a program usually needs to be written in Java language. Fortunately, developers who want to use Hadoop without Java can take advantage of Hadoop Streaming library which allows developers to write MapReduce programs with any language that supports standard input and output. However, Hadoop Streaming suffers from several limitations. At first, the developers can only customize the mapper, combiner and reducer via executable scripts. They cannot provide their own record reader, record writer and partitioner to take more control of Hadoop. Furthermore, HDFS is also not accessible from scripts. Finally, Hadoop Streaming is only appropriate with problems which only need to process text data streams.

4.3 Extend Hadoop Pipes for Standard ML

Hadoop provides a C++ interface called Hadoop Pipes to write MapReduce programs with C++. With Hadoop Pipes, programmers can access and customize various MapReduce components in Hadoop. Furthermore, the application is able to process various types of input data.

C++ interface in Hadoop Pipes is written in a way that make it a SWIG-compatible C++ API. This means that an interface for other languages such as Python, Perl can be generated by SWIG-tool [9]. Currently, SWIG supports scripting languages including Javascript, Perl, PHP, Python, Tcl and Ruby; and non-scripting languages including C#, Common Lisp (CLISP, Allegro CL, CFFI, UFFI), D, Go language, Java, Lua, Modula-3, OCAML, Octave and R. Paper [23] represented a Python package Pydoop that provides

a Python API for Hadoop MapReduce and HDFS. The package is built based on Pipes (C++). Figure 4.1 shows the integration of Pydoop with C++ API. Method calls from MapReduce framework flow through the C++ pipes and Pydoop API and finally calls user-defined methods. The results which are Python object then are wrapped and returned to the framework. To reach HDFS operations, function calls are initiated by Pydoop and parsed to C calls to libhdfs library. Results are then wrapped back into Python objects and returned to the application.

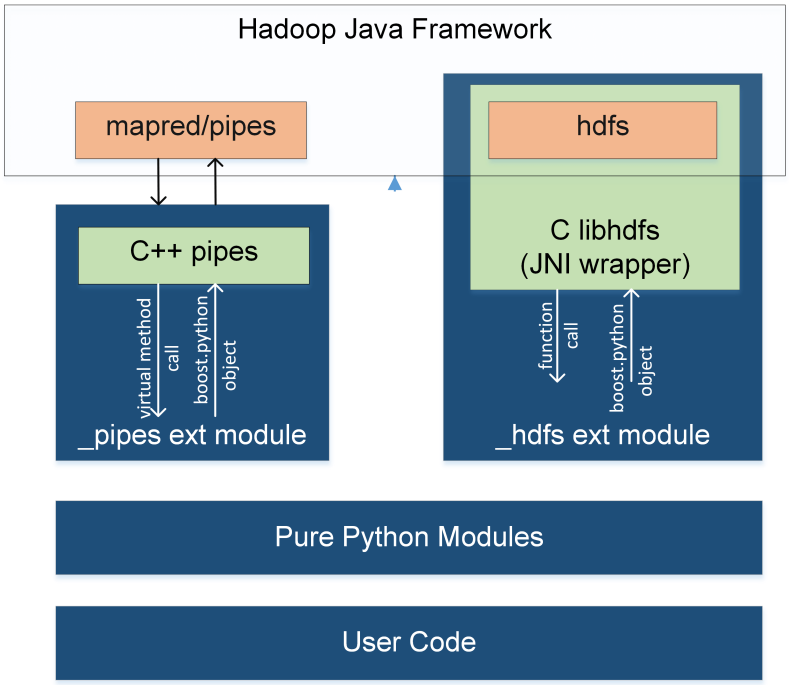

Figure 4.1: Integration of Pydoop with C++

The approach in Pydoop suggests a way to create Hadoop interface for programming languages other than Java. At first, we need to provide APIs for the target language to access MapReduce components in Hadoop. Secondly, the target language and C++ Pipes have to be provided an interface to communicate with each other. In that way, C++ Pipes can delegate the task to the function written in the target language. This requirement can be achieved through SWIG. If the target language supports an interface to communicate with C++, that interface might also be a solution.

Above analysis provides an approach to extend Hadoop Pipes for Standard ML. If we use MLton as a compiler for SML, we have an interface to communicate between SML and C/C++: MLton's Foreign Function Interface (MLton's FFI) [4]. It extends Standard ML and allows SML code to take the address of C global objects, access C global variables. Especially, it allows us to call from SML to C and call from C to SML.

4.4 Architecture of MLoop

The library which is developed in this thesis is named MLoop, which stands for Standard ML API for Hadoop. As described in previous section, developing MLoop is the task

of providing Standard ML API to write mapper, combiner and reducer for Hadoop job. Moreover, MLoop is also destined for allowing customization in record reader and writer. Figure 4.2 shows the architecture of MLoop.

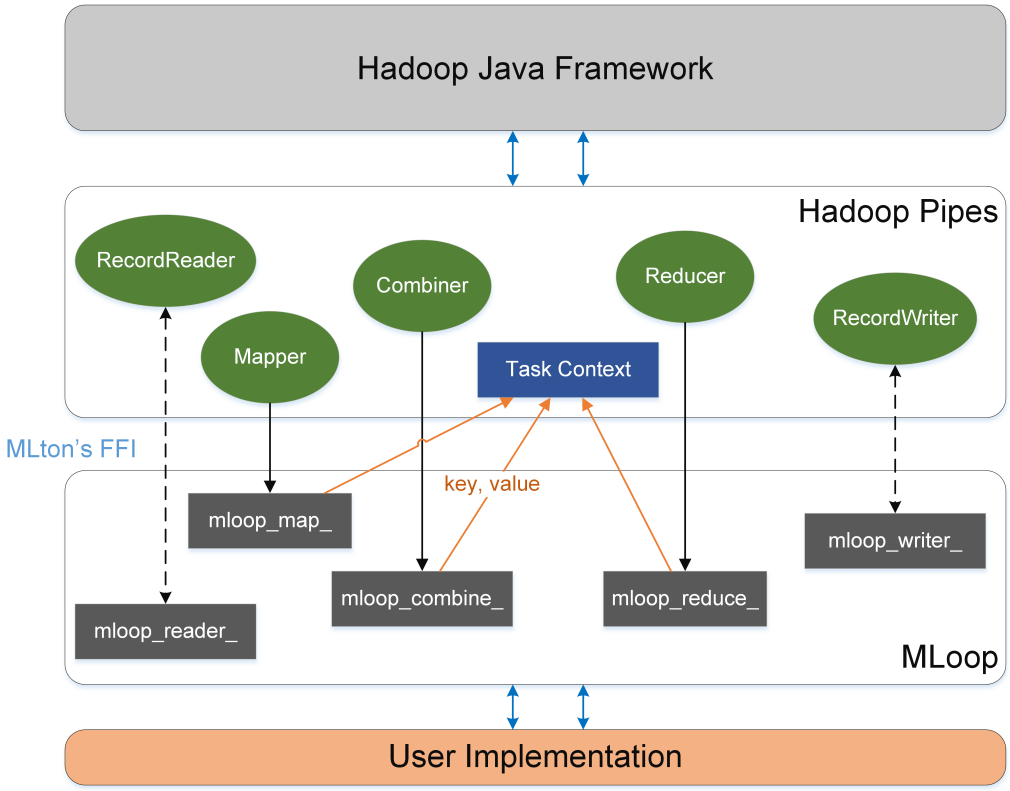

Figure 4.2: SML API for Hadoop

The MapReduce tasks are delegated from Hadoop Java Framework to corresponding C++ components in Hadoop Pipes. These components, in turn, call the user implementations which are provided by MLoop to actually do the tasks. The provided implementations simply delegate the tasks to use code in Standard ML through MLoop library. In detail, C++ classes which are responsible for the tasks call corresponding functions in MLoop to accomplish the tasks. Mapper, Combiner and Reducer objects call `mloop_map_`, `mloop_combine_` and `mloop_reduce_` functions in MLoop, respectively. The key-value outputs of these functions are reported back to Hadoop Java Framework through the Task Context object in Hadoop Pipes. On the other hand, RecordReader object communicates with `mloop_reader_` to get the input key-value pair and itself reports the result to Hadoop Framework. RecordWriter object works the same way with RecordReader object. It calls `mloop_write_` to get the custom final key-value pair and reports back to the Hadoop Framework. We use MLton compiler for Standard ML because it generates efficient executables. Hence, the communications between Hadoop Pipes objects and MLoop functions are carried out with MLton's FFI. We should notice that functions in MLoop which are called by Hadoop Pipes objects actually do nothing. They instead call user implementations in SML to accomplish the tasks.

4.5 MLoop Implementation

In the implementation, we have to deal with two main issues. The first issue is the communication between different languages. Hadoop Pipes written in C++ while Standard ML can only communicate with C. The second issue is the data-flow of MLoop have to conform with the operations in Hadoop Pipes so that the overall process can run as smoothly as possible.

4.5.1 Communication between SML, C and C++

Hadoop Pipes is written in C++. Therefore, to make MLoop work with Hadoop Pipes, we need to communicate between C and C++ and between SML and C. Fortunately, all of these communications are already supported.

Communicate between C and C++

The situation is that we have written a program in C and we need to integrate an existing third party C++ library into our program. We have a C++ class as shown in Table 4.1.

MyClass.hpp	MyClass.cpp

`#ifndef __MYCLASS_H`
`#define __MYCLASS_H`
 	
`class MyClass {`
`private:`
`  int m_i;`
`public:`
`  void int_set(int i);`
`  int int_get();`
`};`
 	
`#endif`
	`#include <MyClass.h>`
	`void MyClass::int_set(int i) {`
	`  m_i = i;`
	`}`
	
	`int MyClass::int_get() {`
	`  return m_i;`
	`}`

Table 4.1: A C++ class

To call C++ function above, we have to solve two problems: the name mangling of C is different from C++ and C doesn't know classes. To solve these problem, we write a C-wrapper of C++ class as shown in Table 4.2.

Now we can call C++ function in our C code.

```
#include "MyWrapper.h"
#include <stdio.h>

int main(int argc, char* argv[]) {
  struct MyClass* c = newMyClass();
  MyClass_int_set(c, 3);
  printf("%i\n", MyClass_int_get(c));
  deleteMyClass(c);
}
```

Communicate between C and SML

As mentioned before, we use MLton compiler for Standard ML. MLton provides an interface called Foreign Function Interface which allows a SML program to import C functions.

MyWrapper.h

```
#ifndef __MYWRAPPER_H
#define __MYWRAPPER_H

#ifdef __cplusplus
extern "C" {
#endif

  typedef struct MyClass MyClass;

  MyClass* newMyClass();

  void MyClass_int_set(MyClass* v,
      int i);

  int MyClass_int_get(MyClass* v);

  void deleteMyClass(MyClass* v);

#ifdef __cplusplus
}
#endif
#endif
```

MyWrapper.c

```
#include "MyClass.h"
#include "MyWrapper.h"

extern "C" {
  MyClass* newMyClass() {
    return new MyClass();
  }

  void MyClass_int_set(MyClass* v,
      int i) {
    v->int_set(i);
  }

  int MyClass_int_get(MyClass* v) {
    return v->int_get();
  }

  void deleteMyClass(MyClass* v) {
    delete v;
  }
}
```

Table 4.2: C Wrapper for MyClass class

Suppose a C function with prototype `int foo(double d, char c);`. MLton extends the syntax of SML to allow expressions like

```
_import "foo": real * char -> int;
```

Therefore, in SML program, we can call C function as following:

```
val foo_ = _import "foo": real * char -> int;
val c = foo_ 2.0 #"c"
val _ = print (Int.toString c)
```

To compile, we use

```
mlton -default-ann 'allowFFI true' -export-header export.h \
   import.sml foo.c
```

where foo.c contains the definition of foo function in C and import.sml contains the SML code above.

MLton's FFI also allows programs to export SML functions to be called from C. A function of type `real * char -> int` can be exported via following declarations:

```
val e = _export "foo": (real * char -> int) -> unit;
val _ = e (fn (x, c) => 13 + Real.floor x + Char.ord c)
```

Then following command will generate a C header file (named export.h) that can be used to call from a C program.

```
mlton -default-ann 'allowFFI true' -export-header export.h \
 -stop tc export.sml
```

4.5.2 MLoop Data-flow

The data-flow of Hadoop Pipes and the architecture of MLoop which are described before have pointed out the requirements of the implementation. It requires C++ implementation classes which are initialized by the Pipes C++ library to handle the map task and reduce task. Besides that, it also requires C wrapper classes for calling C functions from SML.

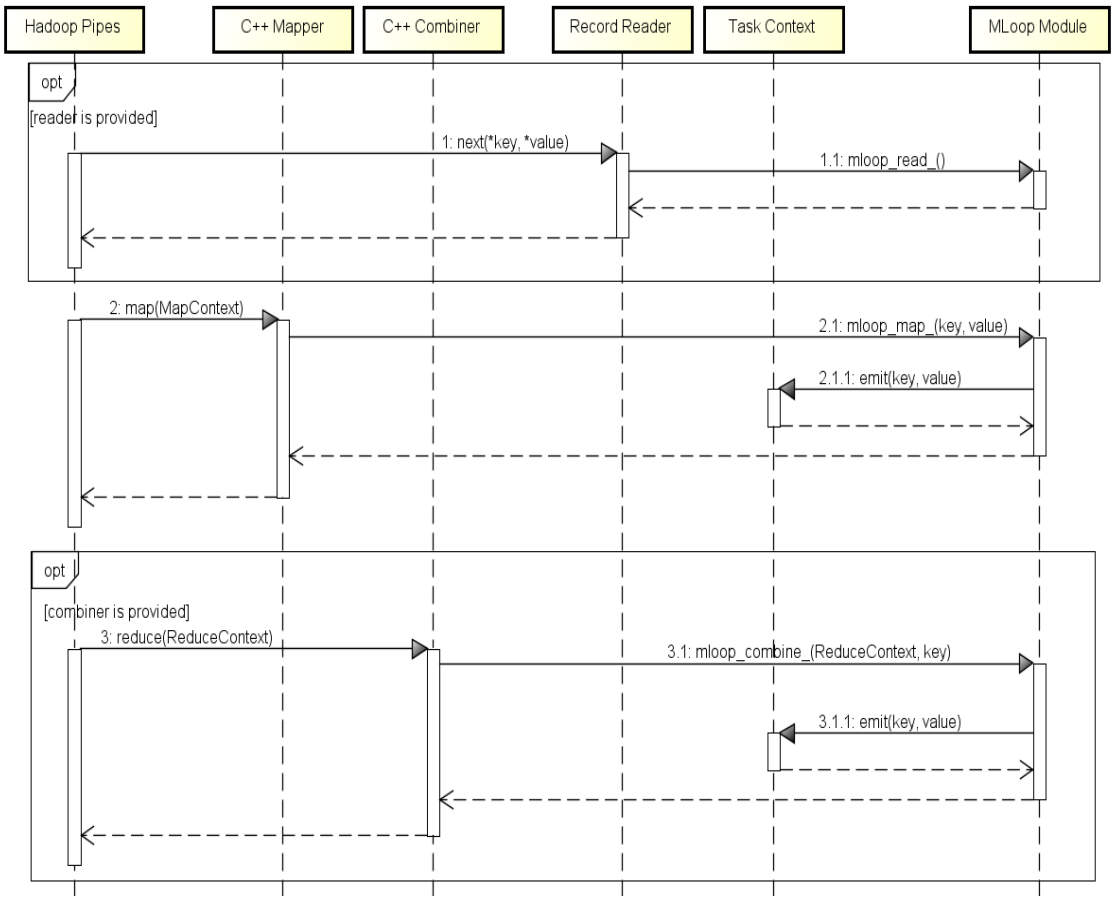

Figure 4.3: Sequence diagram of map phase in MLoop

At least two implementation classes are required to make Hadoop Pipes work: Mapper and Reducer. The other classes provide developers additional controls on operating Hadoop jobs: Combiner helps to reduce the intermediate key-value pairs of map phase, Record Reader controls the way input key-value pairs are generated and fed to mapper, Record Writer deals with storing output pairs of reduce phase. These classes are implemented in MLoop to act as delegates which call MLoop functions to actually do the job.

Figure 4.3 and 4.4 shows the flows inside the MLoop. In the map phase, the Hadoop Pipes object sends a message to `next` function of C++ `RecordReader` implementation (step 1) to read the input file and get an input key-value pair to feed to the `map` function. This only happens if the `RecordReader` object is provided. This object simply forwards the request to `mloop_read_` function of MLoop to get the result key-value pair. If `RecordReader` object is not provided, the system uses the provided Java Record Reader

to parse key-value pairs from the files and feed to the `map` function.

In the next step, the Hadoop Pipes object sends a `MapContext` object which contains the input key-value pair to `map` function of C++ `Mapper` implementation (step 2). The request then is forwarded to `mloop_map_` of MLoop to invoke the user code in SML which contains the business logic of map step (step 2.1). The output key-value pairs are emitted back to Hadoop Pipes via a calls to `TaskContext` object provided by Hadoop Pipes (step 2.1.1).

If the combiner is provided, the intermediate key-value pairs are buffered. At the end of map phase, these pairs are fed to `Combiner` object (step 3). It, in turn, calls `mloop_combine_` of MLoop to do the job (step 3.1). Again this function only takes responsibility to call the user code to do the business logic. The new output key-value pairs are then reported to Hadoop Java Framework via `TaskContext` object (step 3.1.1). If there is no combiner, the intermediate key-value pairs are reported directly to Hadoop Java Framework each time a new pair arrives to `TaskContext` object.

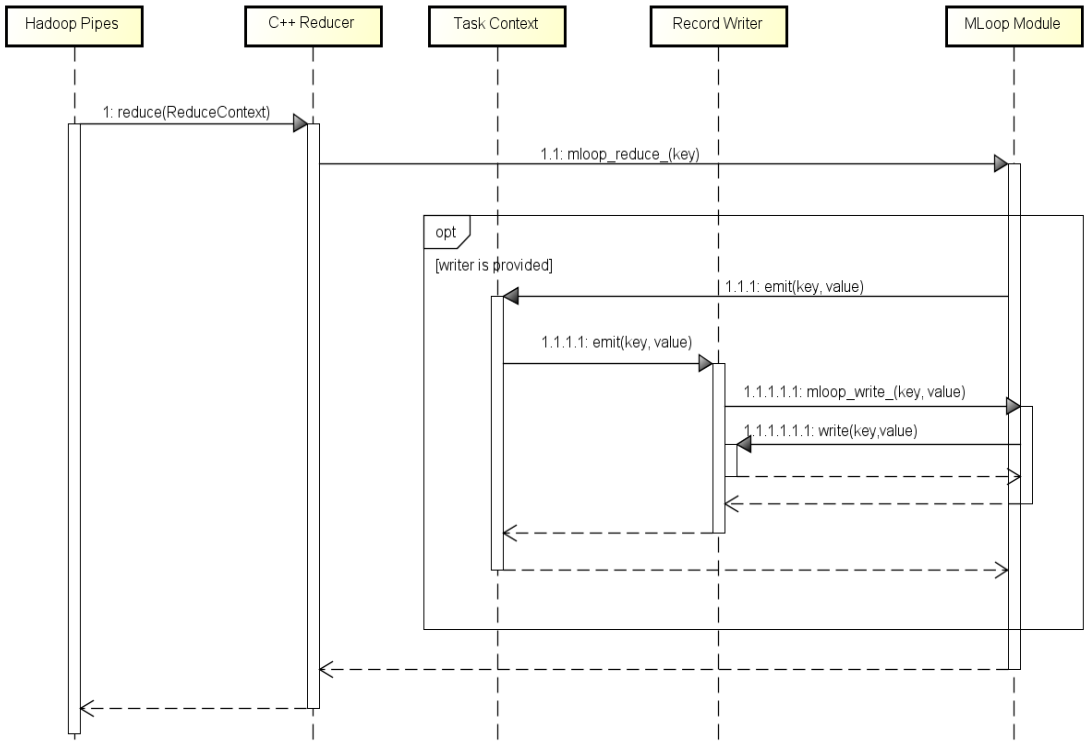

Figure 4.4: Sequence diagram of reduce phase in MLoop

In the reduce phase (Figure 4.4), `reduce` function of C++ `Reducer` is passed a `ReduceContext` object from which the key and all the values associated with it could be retrieved (step 1). The input key is sent to `mloop_reduce_` function of MLoop (step 1.1) and then sent to user-implemented function `mloop_reduce`. This key together with functions provided by MLoop to get associated values are enough to process the actual business logic. The output key-value pairs are returned to `TaskContext` object to write back to file system (step 1.1.1). At this point, there are two cases which can happen. If the writer is provided, the key-value pairs are sent to the writer. It delegates these pairs to `mloop_write_` (step 1.1.1.1.1) to allow user to customize the final key-value pairs.

These final pairs are written directly to HDFS by the writer. On the other hand, if there is no writer, the output key-value pairs from reducer are sent back to Java Framework to write to file system using Java Record Writer.

4.5.3 MLoop Implementation Details

This part reports technical implementation of MLoop based on the architecture and data-flow described in previous sections. MLoop is considered as a channel between the user code in SML and Hadoop Pipes framework in C++. Therefore, MLoop must provide two counterparts: SML module to communicate with user code in SML, provide structures, functions that support user code; C++ module to retrieve the task from Hadoop Pipes framework, forward to SML module, receive the result from user code and report to Pipes framework.

Standard ML API for Hadoop

MLoop aims to provide a very simple API for user to write the MapReduce program in Hadoop. Indeed, to control different aspects of MapReduce job, user only needs to implement following functions:

```
fun mloop_map (key:string, value:string):unit
fun mloop_reduce (key:string):unit
fun mloop_combine (key:string):unit
fun mloop_read ():bool
fun mloop_write (key:string, value:string):unit
```

`mloop_map` allows user to define the behavior of map function of MapReduce model. It takes two strings as key-value pair and emits intermediate key-value pair(s). MLoop provides `MapContext` structure to support user to manipulate the map task. `MapContext` provides function `emit: string * string -> unit` to emit the intermediate pair.

`mloop_reduce` and `mloop_combine` allow user to define the bahavior of the reduce and combiner function of MapReduce model. Input key is given as a parameter of the function. MLoop contains `ReduceContext` which provides functions that are needed to retrieve necessary information: `getValueSet: unit -> string list` gets all associated values of the key at a time, `nextValue: unit -> bool` checks if there is still another value associated with current key and `getInputValue: unit -> string` actually gets the associated value. Finally, function `emit: string * string -> unit` is used to emit the output key-value pair.

`mloop_read` and `mloop_write` allow user to customize the record reader and writer. `mloop_write` takes key-value pair from reduce step as inputs. It may generate new key-value pair to actually be written to the file system. Function `emit: string * string -> unit` in `Writer` structure is defined to do that. `mloop_read` uses functions provided by `Reader` structure to parse input split to key-value pairs which are fed to map function. `mloop_read` returns a *true* value indicating that there is still other pairs. Otherwise, it returns *false*. Supporting functions in `Reader` structure is listed as below.

getOffset ()
: return the offset of the current input split which corresponds to this map task.

getBytes_read ()
: return the number of bytes consumed by the reader for the last read operation.

updateOffset_bytesConsumed (offset, bytesConsumed)
: update the current offset of the reader in the file and the number of bytes consumed for the last read operation.

getOffsetNow ()
: return the current offset of the reader in the file.

get ()
: return the reader.

seekHdfs offset
: seek to given offset in file.

The detailed description of all structures which are provided by MLoop is shown in Appendix A.3.

The C++ Mapper

MLoop needs to provide a C++ implementation of Hadoop Pipes `Mapper` class to delegate the task to user-implemented function `mloop_map`. Listing 4.1 shows a partial view of the C++ Mapper provided by MLoop. When the object is created, it sends the address of `MapContext` C++ object to `MapContext` structure (line 13) to store it because `emit` function of `MapContext` structure needs this information to retrieve the original MapContext C++ object to actually emit the output (key, value) pair. When the map function of MLoopMapper is called from Hadoop Pipes Framework, it retrieves the key and value from MapContext object and forwards them to `mloop_map_` (line 18). The responsibility of `mloop_map_` is to convert C++ string pointers to SML strings and feed them to `mloop_map` function. Listing 4.2 shows the definitions of functions in SML module of MLoop.

Listing 4.1: C++ Mapper implementation

```
1  class MloopMapper : public hp::Mapper {
2  private:
3    char* addr;
4  public:
5    MloopMapper(hp::MapContext& ctx);
6    void map(hp::MapContext& ctx);
7    void close(){}
8    virtual ~MloopMapper(){}
9  };
10
11 MloopMapper::MloopMapper(hp::MapContext& ctx) {
12   hp::MapContext* ptr = &ctx;
13   init_map((CPointer) ptr);
14 }
15 void MloopMapper::map(hp::MapContext& ctx) {
16   string key = ctx.getInputKey();
17   string value = ctx.getInputValue();
18   mloop_map_((CPointer) key.c_str(), (CPointer) value.c_str());
19 }
```

Listing 4.2: MLoop setup for map task

```
  fun init_map addr = MapContext.setAddress addr
```

```
fun mloop_map_ (key:MLton.Pointer.t, value:MLton.Pointer.t) = mloop_map (
    fetchCString key, fetchCString value)
```

The C++ Reducer and Combiner

C++ Reducer and Combiner implementations (MloopReducer and MloopCombiner) are also provided by MLoop (Listing 4.3). The address of C++ ReduceContext object is stored in `ReduceContext` structure each time a MloopReducer object is created (line 18). `reduce` function of MloopReducer gets the input key from C++ ReduceContext object and sends to `mloop_reducer` via `mloop_reduce_` (line 22-23). `reduce` function of MloopCombiner works with a slightly different manner. It sends also the address of C++ ReduceContext object. This is because many ReduceContext objects are created to combiner different chunks of intermediate (key, value) pairs. Listing 4.4 shows the counterparts of MloopReducer and MloopCombiner in SML module of MLoop.

Listing 4.3: C++ Reducer and Combiner implementation

```
1  class MloopReducer : public hp::Reducer {
2  public:
3      MloopReducer(hp::ReduceContext& ctx);
4      virtual void reduce(hp::ReduceContext& ctx);
5      virtual void close();
6      virtual ~MloopReducer();
7  };
8
9  class MloopCombiner : public MloopReducer {
10 public:
11     MloopCombiner(hp::MapContext& ctx);
12     virtual void reduce(hp::ReduceContext& ctx);
13     virtual void close();
14 };
15
16 MloopReducer::MloopReducer(hp::ReduceContext& ctx) {
17     hp::ReduceContext* ptr = &ctx;
18     init_reduce((CPointer) ptr);
19 }
20
21 void MloopReducer::reduce(hp::ReduceContext& ctx) {
22     std::string key = ctx.getInputKey();
23     mloop_reduce_((CPointer) key.c_str());
24 }
25
26 void MloopCombiner::reduce(hp::ReduceContext& ctx) {
27     hp::ReduceContext* ptr = &ctx;
28     std::string key = ctx.getInputKey();
29     mloop_combine_((CPointer) ptr, (CPointer) key.c_str());
30 }
```

Listing 4.4: MLoop setup for reduce and combiner task

```
fun init_reduce (addr:MLton.Pointer.t) = ReduceContext.setAddress addr
fun mloop_combine_ (address:MLton.Pointer.t, key:MLton.Pointer.t) =
    let
        val _ = ReduceContext.setAddress address
    in
        mloop_combine (fetchCString (key))
```

```
    end
fun mloop_reduce_ (key:MLton.Pointer.t) = mloop_reduce (fetchCString (key))
```

The C++ Record Reader

Listing 4.5 and 4.6 show the main view of Reader in MLoop. MloopRecordReader parses information about its corresponding input split (line 2-11). This information is stored in `Reader` structure through calling to `reader_init` function (line 20). This information is also used to connect directly to HDFS (line 13-15) and create a LineReader object to read each line of input split (line 18). The Hadoop Pipes framework calls `next` function of MloopRecordReader to get input key-value pair for map function. `next` function, in turn, delegates to user-implemented function in SML API, `reader_read` to response to the call from Hadoop Pipes (line 25).If the result is yes (a positive number), the passed key-value pair is already stored in two class variables *key*, *value*. They are returned to Hadoop Pipes via two reference variables *key_*, *value_* (line 26-30).

Listing 4.5: C++ Record Reader implementation

```
1  MloopRecordReader::MloopRecordReader(hp::MapContext& ctx) {
2      char* home = getenv("HADOOP_HOME");
3      setenv("CLASSPATH", get_hadoop_classpath(home), 1);
4      key = value = NULL;
5      string split = ctx.getInputSplit();
6      _StringInStream is(split);
7      string filename;
8      hu::deserializeString(filename, is);
9      offset = is.readLong();
10     length = is.readLong();
11     int pos = filename.find('/', 7);
12
13     fs = hdfsConnect("default", 0);
14     file = hdfsOpenFile(fs, filename.substr(pos).c_str(), O_RDONLY,
15             0, 0, 0);
16     bytes_read = 0;
17     start = offset;
18     in = new LineReader(fs, file);
19     newLine = NULL;
20     reader_init((CPointer) this, (Int64) offset, (Int64) length);
21     reader_setup();
22 }
23
24 bool MloopRecordReader::next(std::string& key_, std::string& value_) {
25     int32_t res = reader_nextVal();
26     if (res) {
27         key_ = *key;
28         value_ = *value;
29         return true;
30     }
31     return false;
32 }
```

Listing 4.6: SML functions for reader

```
fun reader_init(file,offset,length) =  Reader.init (file,offset,length)
fun reader_setup () = let
    val offset = Reader.getOffset()
```

```
      val reader = Reader.get()
      in
          if offset = 0 then ()
          else (seekHdfs(reader, offset - 1); Reader.readLine ();())
      end
  fun reader_nextVal () = reader_read ()
```

The C++ Record Writer

The Record Writer controls the way that output key-value pairs are written to the file in HDFS. The current implementation of Writer in MLoop allows users to customize the output key-value pair as well as define the separator between key and value. The MlooopRecordWriter works directly with HDFS to create output file (2-17). The address of current writer object is stored in `Writer` structure to access it later. Whenever a key-value pair needs to write to the file system, the writer object invokes user-implemented function `mloop_write` to process (line 22). Listing 4.7 and 4.8 show the implementation.

Listing 4.7: C++ Record Writer implementation

```
1 MloopRecordWriter::MloopRecordWriter(hp::ReduceContext& ctx) {
2     char* home = getenv("HADOOP_HOME");
3     setenv("CLASSPATH", get_hadoop_classpath(home), 1);
4     const JobConf* conf = ctx.getJobConf();
5     string out_dir = conf->get("mapreduce.output.fileoutputformat.outputdir
          ");
6     int pos = out_dir.find('/', 7);
7     out_dir = out_dir.substr(pos);
8     int part = conf->getInt("mapred.task.partition");
9     char s[200];
10    sprintf(s, "%s/part-%.5d", out_dir.c_str(), part);
11    fs = hdfsConnect("default", 0);
12    file = hdfsOpenFile(fs, s, O_WRONLY | O_CREAT,
13            0, 0, 0);
14    string sep("\t");
15    if (conf->hasKey("mapred.textoutputformat.separator"))
16        sep = conf->get("mapred.textoutputformat.separator");
17    writer = new LineWriter(fs, file, sep);
18    writer_init((CPointer) this);
19 }
20
21 void MloopRecordWriter::emit(const std::string& key, const std::string&
      value) {
22    mloop_write_((CPointer) key.c_str(), (CPointer) value.c_str());
23 }
```

Listing 4.8: SML functions for writer

```
  fun writer_init address =  Writer.store address
  fun mloop_write_ (key,value) = mloop_write (key,value)
```

4.6 Writing MapReduce programs in MLoop

4.6.1 MLoop Basics

Writing MapReduce programs in MLoop is similar to writing in Hadoop Java. We just need to define two functions `mloop_map` and `mloop_reduce`. We are also able to provide

the combiner via function `mloop_combine`.

Listing 4.9: Counting words in MLoop

```
1  fun mloop_map (key:string, value:string) =
2      let
3          val splitter = String.tokens(fn c => Char.isSpace c)
4          val words = splitter value
5
6      in
7          map (fn word => MapContext.emit (word,"1")) words
8      end
9
10 fun mloop_reduce (key:string)=
11     let
12         fun fromString str = let
13             val x = Int.fromString str
14             in
15                 Option.getOpt(x,0)
16             end
17         val sum = ref 0
18     in
19         while (ReduceContext.nextValue()) do
20             sum := (!sum) + (fromString (ReduceContext.getInputValue()));
21         ReduceContext.emit(key, Int.toString(!sum))
22     end
23
24 fun mloop_combine (key:string) =
25     mloop_reduce (key)
26
27 val useCombiner = true
```

Listing 4.9 shows the Word Count program in MLoop. In the map function, the *value* argument contains a line of input. In line 3, we use function `tokens` of structure `String` which takes two arguments (f, s) to get a list of tokens derived from string `s` by a delimiter which is a character satisfying the predicate `f`. In our code, `f` is the function `Char.isSpace` which retrieves a character `c` and returns true if `c` is a whitespace (space, newline, tab, carriage return, vertical tab, form-feed). We only pass a predicate f to function `tokens`. Therefore, it returns a another function with type `string -> string list`. We then apply that function which is bound to variable `splitter` to the string `value` to tokenize the input line with whitespace delimiter (line 4). In line 7, we use the `map` function of Standard ML,which is used to apply a function f on every element of a list, to emit each token as *key* with associated count of "1".

In reduce function (`mloop_reduce`), we first define a function `fromString` to convert the input string into number (line 12-16). We also initialize a reference cell which is pointed to by `sum` to keep track of the total occurrences of a word. We then loop over all the associated values of the input key to sum all the counts (line 19-20). Finally, we emit the output key-value pair to the Hadoop Pipes framework (line 21).

We also provide a combiner for this job which is the same as the reduce function (line 24-25). Finally, we need to notify the system that we want to use combiner feature. That is specified by setting variable `useCombiner` to `true`.

4.6.2 Record Reader

By default, Hadoop splits input data into text lines. The key-value pairs are generated from these lines. The *key* is the byte offset within the file of the beginning of a line. The *value* is the content of the line. If you want to process multiple line at a time, you need to write a custom Record Reader.

The Record Reader works at the HDFS file level: it reads data from the file, generates key-value pairs to pass into the Mapper. To write a Record Reader in MLoop, two functions must be defined: `setUpReader` and `mloop_read`.

Listing 4.10: A custome record reader to read two lines at a time

```
1  (* define map and reduce here *)
2
3  fun setUpReader () = let
4      val offset = Reader.getOffset()
5      in
6          if offset = 0 then ()
7          else (Reader.seekHdfs(offset - 1); Reader.readLine ();())
8      end
9
10 fun mloop_read () = let
11     val start = Reader.getOffsetNow ()
12     val endOffset = (Reader.length()) + (Reader.getOffset())
13     val (value,isSuccess) = if start < endOffset then (Reader.readLine (),
           true) else ("",false)
14     val (value2, isNext) = if isSuccess andalso (Reader.getOffsetNow ()) <
           endOffset then (value ^ (Reader.readLine ()), true)
15         else (value, false)
16     in
17         if isSuccess orelse isNext then setKeyValue (Reader.get(),cstring (
               Int64.toString start),cstring value2)
18         else ();
19         isNext
20     end
21 val useReader = true
```

The reader specified in Listing 4.10 reads the input split into lines, combines two lines into one line. In the setup phase, we get the offset of input split in the file (line 4). If the input split starts in the middle of file, it may start in the middle of line. Therefore, we remove a line which starts right before the offset of current input split (line 6-7). This makes sure that we do not read again the last line of previous input split. The actual task of reading key-value pair takes place in `mloop_read` function. At first, the current offset of reader and the last offset of input split are stored into `start` and `endOffset`, respectively (line 11-12). The reader tries to read a line from current offset. If it succeeds, `value` stores the line and `isSuccess` is set to `true`. Otherwise, `value` is an empty string and `isSuccess` is set to `false` (line 13). In line 14, the reader tries to read the second line if previous operation is successful and the current offset is still in the input split. If this operation is successful, the new line is append to old line `value`. Finally, we need to send the key (start offset) and the value (lines which are read) to MloopRecordReader object via `setKeyValue` which updates the class variables *key* and *value* of MloopRecordReader object (line 17-18). The function returns the boolean value `isNext` which indicates that there is maybe value for current input split to read. In line 21, we notify that we want to use Record Reader to read input split.

4.6.3 Record Writer

The Record Writer controls the way that output key-value pairs are written to the file in HDFS.

Listing 4.11: A custom record writer in MLoop

```
(* define map and reduce here *)
fun mloop_write (key,value) = Writer.emit (key, value ^ "|")
val useWriter = true
```

The writer specified in Listing 4.11 changes the value by appending to it a vertical bar "|". The key and new value is sent to MloopRecordWriter to write directly to output file in HDFS using `Writer.emit`.

4.7 Sample Programs in MLoop

In chapter 5, we want to compare the performance between different large-scale parallel solutions on different problems. Therefore, we again solve the same problems described in section 2.3, but use MLoop instead. The complete source code can be found in Appendix section B.2.

4.7.1 Summation

Working with calculating the sum of whole numbers from one to ten billion requires dealing with big integer. Fortunately, Standard ML supports it with `IntInf` structure. Operations with big numbers in Standard ML is based on GMP library. Therefore, it is very efficient. Listing 4.12 shows the code for map and reduce function written in MLoop.

Listing 4.12: Summation in MLoop

```
1  fun mloop_map (key:string, value:string) =
2    let
3      val splitter = String.tokens(fn c => Char.isSpace c)
4      val words = splitter value
5      fun toInt number = let
6        val x = IntInf.fromString number
7      in
8        Option.getOpt(x,0)
9      end
10     fun sum (from:IntInf.int,to:IntInf.int, result:IntInf.int) = if from >
           to then result
11       else sum (from + 1, to, result + from);
12     val v = sum (toInt (List.nth(words,0)), toInt (List.nth(words,1)), 0);
13   in
14     MapContext.emit ("", IntInf.toString v)
15   end
16
17 fun mloop_reduce (key:string)=
18   let
19     fun fromString str = let
20       val x = IntInf.fromString str
21     in
22       Option.getOpt(x,0)
23     end
24     val sum = ref 0 : IntInf.int ref
```

```
25   in
26     while (ReduceContext.nextValue()) do
27       sum := (!sum) + (fromString (ReduceContext.getInputValue()));
28     ReduceContext.emit(key, IntInf.toString(!sum))
29 end
```

Each map function calculates the sum of a sequence of numbers: $x, x+1, ..., x+n$. This sequence is represented by two numbers: $x, x+n$. The value argument of `mloop_map` contains these two numbers which is represented in text format and is separated by whitespace delimiter. Therefore, the map function at first tokenize the value to get these two numbers (line 3-4). Function `toInt` is used to convert from string to number. Two passed numbers are put into function `sum` to calculate the sum (line 12). All map functions emit the same key "" and the sum of its sequence as value so that all the partial sums come to a reducer (line 14). The reduce function of this problem is very similar to the reduce function described in Word Count problem, which calculates the sum of all the associated values of the input key (line 17-29).

4.7.2 17-Queens

N-Queens problem can be solved with iterative Map-Reduce. In this example, we try to use MLoop to find the solution for it. The solution starts with trying to find all possible ways of putting first n queens on the board, then finding the complete solution for each way.

Listing 4.13: MapReduce job to put one more queen into the board

```
1 fun addQueen (qs,n) = let
2   val col = 1 + List.length qs
3   fun put row = if row > n then ()
4   else if conflict (col,row) qs then put (row + 1)
5   else (MapContext.emit(randInt (), codeBoard((col,row)::qs, "")); put (row
        + 1))
6  in
7      put 1
8  end
9
10 fun append (row,nil) = [(1,row)]
11   | append (row, (col,x)::t) = (col + 1, row) :: (col,x)::t
12
13 fun parseBoard value = let
14   val splitter = String.tokens(fn c => c = #"-")
15   fun toInt s = Option.getOpt (Int.fromString s, 0)
16   val values = map toInt (splitter value)
17   in foldl append nil values
18 end
19
20 fun mloop_map (key,value) = if value = "START" then addQueen ([],17)
21   else addQueen(parseBoard value,17)
22
23 fun mloop_reduce (key:string)=
24     while (ReduceContext.nextValue()) do
25         ReduceContext.emit("", ReduceContext.getInputValue())
```

The code snippet to find all possible ways to add one more queen on the next column is listed in Listing 4.13. The board is represented as a string. It can be "1-3-5" means the queens were put at the first, third and fifth row on the first, the second and the

third column respectively; or the string can just be "START" means no queen was put on the board. That string then is parsed into a list of coordinates of queens on the board using function `parseBoard` (line 13-18). The coordinate has the form (column, row). For example, the string "1-3-5" is parsed as [(1,1),(2,3), (3,5)]. For a specific board with known size which is filled by queens for several first columns, function `addQueen` tries to put a queen on every row (line 1-8). The legal row is reported as key-value pair: the key is randomized from 1 to 10, the value is the string representing the board (line 5). Remember that all values associated with a key are sent to only one reducer. So the purpose of randomizing key is to send the output values (the boards indeed) to different reducers.

The `mloop_map` function receives a key-value pair whose the value contains the current state of the board. If no queen was put on the board (value equals to "START"), then the program find all possible ways to put one queen on the first column: `addQueen([],17)` (line 20). Here 17 is the size of the board. Otherwise, all possible ways to put one more queen on the next column is searched (line 21).

After filling the board with several first columns, we then use depth-first-search to find all possible solutions correspond to each filled board as shown in Listing 4.14. Function `fillQueen` uses depth-first-search approach to find all complete solution for given board and size (line 1-8). Each time it finds a board, it emits to the Hadoop Pipes framework (line 4). The function `mloop_map` just parses the board (line 16), calls `fillQueen` to get all complete solutions and emits them (line 19). The reduce function do nothing rather than emitting all the associated values which represent the board (line 10-12).

Listing 4.14: MapReduce job to find the complete board with 17 queens

```
1  fun fillQueen (qs,n,col,fc) = let
2    fun put row = if row > n then fc ()
3    else if conflict (col,row) qs then put (row + 1)
4    else if col = n then (MapContext.emit(randInt (), codeBoard((col,row)::qs
         , "")); put (row + 1))
5    else fillQueen((col,row)::qs,n,col+1, fn () => put (row+1))
6  in
7      put 1
8  end
9
10 fun mloop_reduce (key:string)=
11     while (ReduceContext.nextValue()) do
12         ReduceContext.emit("", ReduceContext.getInputValue())
13
14
15 fun mloop_map (key,value) = let
16   val board = parseBoard value
17   val col = 1 + List.length board
18 in
19   fillQueen(board, 17, col, fn () => ())
20 end
```

We define two kinds of job to solve 17-Queens problem. However, MLoop do not support a mechanism to control the job sequence. Therefore, we have to come up the manual solution. We write script to get the first five jobs run in sequence, one by one, the later uses the output of the former. In first five job, we use the MapReduce job defined in Listing 4.13. Finally, we run the job defined in Listing 4.14 on the output of the last running job. In this way, we achieve the result that we need.

4.7.3 Graph Search

Graph search is already described in previous chapters. It could be solved with iterative MapReduce. Using MLoop, we can implement the solution for this problem with SML.

In the map phase, when a node comes in, it colors turns into "BLACK" (means that it was visited) if its current colors is "GRAY" (means that it is going to being visited) (line 6 - 9). At the same time, we also generate new nodes from each adjacent node with incoming node's distance plus 1, gray color and adjacent list NULL (line 7). If node is not gray, we just push it out (line 11).

In the reduce phase, reducer will get a node id as key and all possible associated information as value set. In line 44-45, we loop over this set. At each step of the loop, the associated information is extracted using function `unpackInfo` to get the adjacent list, the distance and the color. At the end of the loop, valid adjacent list, the smallest distance and maximum color are found. Then reducer just push out a new node with old key and new found values (line 46). The code snippet is shown in Listing 4.15.

Listing 4.15: A partial view of Graph Search in MLoop

```
1 fun mloop_map (key,value) = let
2   val (id,(edges,dist,color)) = unpack value
3   fun increaseDist dist = case (Int.fromString dist) of SOME t => Int.
        toString(t+1) | NONE => "Integer.MAX_VALUE"
4   fun parseEdges e = if e = "NULL" then [] else split (e,#",")
5  in
6   if color = "GRAY" then (
7   map (fn k => printNode (k,"NULL",(increaseDist dist),color)) (parseEdges
        edges);
8   printNode (id, edges,dist,"BLACK")
9   )
10    else
11   printNode (id,edges,dist,color)
12 end
13
14 (* info = EDGES|DISTANCE_FROM_SOURCE|COLOR| *)
15 fun unpackInfo info = let
16   val tmp = split (info,#"|")
17   val valid2 = if List.length tmp = 3 then true else raise Fail("Invalid
        info.")
18   val e::d::c::_ = tmp
19 in
20    (e,d,c)
21 end
22
23 fun mloop_reduce (key:string) = let
24   val dist = ref "Integer.MAX_VALUE"
25   val colour = ref "WHITE"
26   val edge = ref "NULL"
27   fun colorInt color = case color of "WHITE" => 0 | "GRAY" => 1 | "BLACK"
        => 2 | _ => ~1
28   fun maxColor (c1, c2) = let
29     val v1 = colorInt c1
30     val v2 = colorInt c2
31     in
32     if v1 < v2 then c2 else c1
33   end
34   fun minDist (d1, d2) = let
```

```
35   val v1 = case (Int.fromString d1) of SOME t => t | NONE => maxInt
36   val v2 = case (Int.fromString d2) of SOME t => t | NONE => maxInt
37   in
38   if v1 < v2 then d1 else d2
39   end
40   fun update (edges:string,distance:string,color:string) = (if not (edges =
         "NULL") then edge:=edges else ();
41              dist:= minDist (!dist, distance);
42              colour := maxColor(!colour,color))
43  in
44     while (ReduceContext.nextValue()) do
45         update (unpackInfo (ReduceContext.getInputValue()));
46     reduceNode (key,!edge,!dist,!colour)
47  end
```

In this problem, we can use the same approach with 17-Queens problem to run iterative MapReduce job. However, as described in Section 2.3.3, to stop the job, we need to use a counter to keep track of the status of graph. That is what we are not provided in MLoop. Therefore, there is no way to know if all the nodes in the graph are visited or not. A solution for this problem is that we just run the job several times on new output. Then we use another job to check if there is "GRAY"node in the graph or not. If not, we stop. Otherwise, we run above job again on the latest output several times and then check again. We repeat that process till we are sure to have the final graph.

4.8 MLoop Limitations

From actual use cases, we find out that MLoop suffers from several shortcomings. They come from the implementation as well as the nature of the approach to develop MLoop.

In this implementation, MLoop does not support partitioner and Hadoop job counters because the available amount of time for implementation is limited. Partitioner decides the reducer to which an intermediate key-value pairs belongs. Lacking of supporting partitioner makes developers impossible to customize this function. In some situation, the performance of MapReduce jobs is increased a lot if using job counters. This cannot be achieved in MLoop because it does not support counters. Finally, access to HDFS is not fully supported in current implementation of MLoop. It is restricted in pre-defined actions of record reader and writer.

The current approach of MLoop has some drawbacks. Because it extends Hadoop Pipes, which does not provide any mechanisms to chain multiple jobs. This is also impossible in MLoop. Therefore, a class of problems which need iterative jobs to solve are not able to be implemented in MLoop. Furthermore, communication between MLoop and Java framework is through Hadoop Pipes. It introduces overhead on data communications which can reduce the performance of the job.

Chapter 5

Evaluation and Results

In order to verify the efficiency and usefulness of MLoop, we conduct various experiments to compare it with other existing large-scale solutions. Considered solutions include Hadoop, Hadoop Streaming and MPI. Two clusters were set up to run these experiments: a small heterogeneous cluster of high-end machines and a bigger homogeneous one of low-end commodity machines. Different worth noting points then are discovered from the results. From them, we suggest several guide-lines (advice) which should be considered when choosing the parallel processing solution to solve the problem.

5.1 Evaluation Metrics

There are many aspects which need to be considered when choosing a solution for parallel processing. Consider all of them is impossible in this thesis. Instead, we just choose several notable aspects as metrics to evaluate chosen approaches. They include:

Performance In this evaluation, the performance is measured by the execution time - the amount of time which the program needs to solve the problem.

Scalability This mentions how the program scales with different sizes of the same problem. When looking at the scalability, there are two aspects which are under consideration. In terms of data, the execution time of the same algorithm should increase linearly with the amount of data. We use ratio per unit (constant of proportionality) value to compare effective of each solution in this aspect. This value is calculated as follows.

ratio per unit $= \frac{\frac{y_t}{y_0}}{\frac{x_t}{x_0}} = \frac{y_t}{y_0}.\frac{x_0}{x_t}$, where y_t is the execution time when the amount of data is x_t; y_0 is the execution time when the amount of data is smallest, x_0. In ideal case, ratio per unit is 1 means that the execution time increases at the same rate that the amount of data does. In actual case, this value is greater than 1 means that the execution time increases faster than the amount of data does. The smaller value of ratio per unit means better scalability.

In terms of resources, with the same amount of data, the execution time of same algorithm should be inversely proportional to the cluster size. In order to compare this property, we calculate the Constant of Proportionality $a = \frac{y_t}{y_0}.\frac{x_t}{x_0}$. In ideal case, a is equal to 1 means that the execution time decreases at the same rate that

Specification	Machine 1	Machine 2	Machine 3
Processor (CPU)	Core i7 - 3610QM 2.2GHz (4 cores, 2 threads/core)	Core 2 Quad Q9450 2.66GHz (4 cores, 1 thread/core)	Core 2 Quad Q9450 2.66GHz (4 cores, 1 thread/core)
Operating System	Ubuntu 15.04	Ubuntu 14.04	Ubuntu 14.04
Memory	4GB RAM	8GB RAM	8GB RAM
Hard Disk	Read 60 MB/s Write 61 MB/s	Read 77 MB/s Write 80 MB/s	Read 98 MB/s Write 111 MB/s
Network BandWidth	936 megabits per second		

Table 5.1: Local cluster specification

the cluster size increases. However, in general, a is greater than 1 means that the execution time decreases slower than the cluster size increases. This is because there is an extra overhead on communication when increasing the cluster size. In this metric, the smaller value also means better scalability.

Fault Tolerance This shows the robustness of the program with errors and faults. These can come from hardware failures or software bugs. In this evaluation, we only view it as "yes" or "no".

Development Effort It is the effort that the developer needs to spend to develop a program that can solve the problem. This is hard to measure. In this evaluation, it is measured simply by the number of code lines in the program, and the level of how hard or easy to maintain that program.

5.2 Experiments

In our experiment, we evaluate the performance of the system under two input parameters. The first is the cluster configuration. We, indeed, compare the performance of two different clusters: one is set up in local environment and the other is set up on Google Cloud Service. The second input parameter is the parallel approach that we use to solve the problem. In the scope of this experiment, we compare different choices to develop programs for large-scale problems. They include MPI (Chapter 3), MLoop (Chapter 4), Hadoop and Hadoop Streaming (Chapter 2).

5.2.1 Experiment Setup

Two clusters are set up to carry out the experiment. The Hadoop and MPI cluster then are set up on these clusters.

The first cluster includes three machines with strong hardwares. These machines differ from CPU, RAM, disks... The detail information is shown in Table 5.1. In total, this cluster has 12 physical cores or 16 logical cores and 20 GB of memory to handle the tasks.

The second cluster is set up based on Google Compute Engine service. Eight virtual machine instances form a homogeneous cluster. Each instance is a n1-standard-1, which has one virtual CPU and 3.75 GB memory. Each instance then is attached to a 50 GB

Specification	Machine 1 - 8
Processor (CPU)	A single hyperthread on a Intel(R) Xeon(R) E5-2670 @ 2.60GHz
Operating System	Ubuntu 14.04
Memory	3.75GB RAM
Storage	50 GB HDD
Hard Disk	Read 127 MB/s Write 71 MB/s
Network BandWidth	1875 megabits per second

Table 5.2: Google cluster specification

Configuration	Cloud Cluster	Local Cluster
Map Container memory	1.5 GB	1.5 GB
Reduce Container Memory	2 GB	2 GB
Maximum memory	3 GB on each node	3 GB on node 1, 7 GB on node 2 and 3
Total memory	24 GB	17 GB
# Parallel mappers	$\lfloor 3/1.5 \rfloor * 8 = 16$	$\lfloor 3/1.5 \rfloor + \lfloor 7/1.5 \rfloor * 2 = 10$
# Parallel reducers	$\lfloor 3/2 \rfloor * 8 = 8$	$\lfloor 3/2 \rfloor + \lfloor 7/2 \rfloor * 2 = 7$

Table 5.3: Hadoop configuration

"standard persistent disk". The detail specification is listed in Table 5.2. In total, this cluster has 8 physical (logical) cores and 30 GB of memory to process the workload.

In terms of software, all the machines in both clusters have installed JDK 1.7.0_76, OpenMPI 1.8.1 and Apache Hadoop 2.2.0. Hadoop is set up on each cluster so that mapper container uses 1.5 GB of memory while reducer container uses 2 GB of memory. The maximum memory that Hadoop YARN can utilize on each node is set up based on the memory of that node. Table 5.3 shows the detail information.

To evaluate the performance of the cluster, two kinds of applications were tested. We use one data-intensive application - Word Count - and three processor-intensive applications - Graph Search, Summation and 17-Queens problem. However, Graph Search is also considered as a data-intensive problem because its input is not small at all. In each problem, the input files are stored in Hadoop Distributed File System (HDFS). They are also stored in the master node of MPI cluster so that the master process can read and then distribute the data to worker processes. MPI task is tested with different number of processes which are the multiple of the number of logical cores in cluster.

The input data for the Word Count problem is extracted from Amazon movie review dataset [24]. This dataset consists of movie reviews from Amazon. The data were collected during a period of more than ten years, including about eight million reviews. From this dataset, 2GB, 4GB and 8GB are extracted as input files for the Word Count problem. With Hadoop family, we use 6 reducers to generate the result.

The Graph Search problem finds the smallest distance from a given node to every node in the graph. The input for this problem is a huge graph which is generated randomly by a Java program. This graph contains one hundred thousand nodes; each has 1000 adjacent nodes. This graph is stored as adjacency list in a 592 MB text file. The detail format

of input and the algorithm of solution already mentioned in previous chapters. In this experiment, 6 reducers are also used to generate the output in Hadoop cluster.

In Summation problem, the sum from zero to 10^{10} is calculated in a very normal and in-efficient fashion: aggregate the sum one by one from the first number to the last. The sequence is split into ten sub-sequences of equal length to calculate in parallel. The purpose here is to measure the efficiency of parallel processing with Hadoop and MPI.

Another processor-intensive problem which is considered here is the 17-Queens. This is the problem of placing 17 chess queens on a 17x17 chessboard so that no two queens threaten each other. There are different optimized solutions for this. However, the purpose of this experiment is to evaluate the performance of parallel solutions. Therefore, the brute-force solution is applied to solve this problem. The detail algorithm is also described in previous chapters. In this experiment, Hadoop cluster is set up to run with 8 reducers.

5.2.2 Experimental Results

Experiments were conducted to compare different large-scale parallel processing solutions based on several pre-defined evaluation metrics. Therefore, the experiment results which draw from these experiments will be discussed separately on each of chosen metric. To get the performance result, each evaluation is conducted three times. The figures which are reported are the average values.

Performance

In this metric, the performance differences between different solutions and between two setup of clusters are analyzed. Furthermore, from the result of related work, we know that MPI provides the best result and among the Hadoop family, the native Hadoop is the best. However, we want to know how much better MPI is and to know the difference in performance of our MLoop library and the native Hadoop. Therefore, these two concerns are especially focused. Furthermore, in some experiments, we will see the appearance of MLoop with combiner enabled (programs written with MLoop, using combiner to optimize the performance) but not Hadoop Streaming with combiner enabled. That is because we only want to know the effect of using combiner in MLoop library.

Figure 5.1 shows the execution time on Word Count when input file size is 2.1 GB. It is easy to see that MPI out-performs other solutions. On the local cluster, the MPI job only needs 50.6% the amount of time which the best of remaining solutions - native Hadoop - needs. It means that MPI performs at least $1/0.506 = 1.98$ times faster. On Google cloud cluster, the MPI job spends more time. However, it is still at least 1.21 times faster than others. This can be explained. MPI job on cloud cluster has less cores to handle the tasks than the one in local cluster (8 cores vs 16 cores). Therefore, the performance of MPI on cloud cluster is worse than on local one.

Among the family of Hadoop, the native Hadoop - in which programs are written in Java - provides the best performance while Hadoop Streaming is the worst one. We can see that the performance of Hadoop family on Google cloud cluster is better than local cluster: 235.7 seconds compared with 273.3 seconds. The reason is simple. Hadoop divides the job into smaller tasks. The number of parallel tasks that could be run at the same time depends on the amount of available memory so that YARN resource manager can allocate resource for the container. Table 5.3 shows that the cloud cluster has bigger amount of available memory: 24 GB in comparison with 17 GB. However, we could see

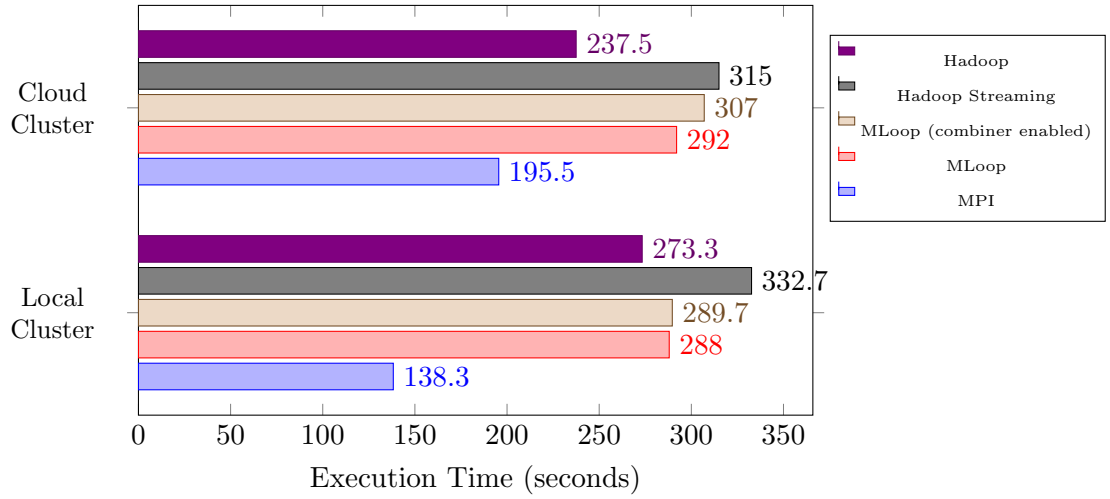

Figure 5.1: Performance on Word Count problem

that the judgment which we have with MPI above is not true with MLoop. We already know that increasing the degree of parallelization also increases communication costs. One of disadvantages of MLoop architecture is that it uses a lot of communications between modules. This can cause overhead on the performance as we see: MLoop performs worse on cloud cluster.

There are two versions of MLoop which is tested in this problem: with and without the combiner. With Word Count problem, the combiner helps to improve the performance a lot, at least in theory. However, MLoop with combiner enabled does not work better than the one without combiner. Enabling combiner increase the execution time from 292 seconds to 307 seconds and 288 seconds to 289.7 seconds on cloud and local cluster, respectively. The overhead of enabling combiner in MLoop may be too big in this case. An interesting point from the result is that MLoop works better than Hadoop Streaming: the execution time is 6.3% less in cloud cluster (292 seconds versus 315 seconds) and 13.4% less in local cluster (288 seconds compared with 332.7 seconds). In comparison with the native Hadoop, MLoop provides promising results. On cloud cluster, the performance of MLoop is as 81.3% (237.5/292) as the performance of the native Hadoop. This ratio is even much better on local cluster when it is very close to 1, about 94.5% (273.3/288).

The results of Graph Search problem are shown in Figure 5.2. Note that in this experiment, the reported execution time of MLoop and Hadoop Streaming is just the amount of time to produce the answer in case we know the height of the tree. Otherwise, the execution time must be greater than reported figures because we need more time to check if that is the final answer. The purpose of this experiment is to evaluate the performance when chaining multiple jobs. Therefore, it's reasonable to have the assumption which we know the height of the tree. The best solution is still MPI. It saves about 28% (211.4 seconds compared with 292.3 seconds on local cluster and 303.9 seconds compared with 422.3 seconds on cloud cluster) the amount of time which the best one on remaining solutions has to spend to solve the problem. The native Hadoop is still the best choice in Hadoop family. The notable point in this problem is that MLoop really offers better results compared with Hadoop Streaming. On cloud cluster, it saves the developer 4% the amount of time when only needing 490 seconds instead of 508.7 seconds to finish the task. On local

cluster, the amount of time which is saved is even much more with 8% degradation from 343.3 seconds to 314.3 seconds. This is brought back with the use of combiner. It reduces the execution time from 333.3 seconds to 314.3 seconds, a 6% reduction. However, the efficiency of using combiner in this problem is hard to know because the combiner only helps to reduce one seconds (491 seconds and 490 seconds with and without using combiner respectively) on cloud cluster. The result also shows that MLoop is worth to consider as an alternative for the native Hadoop. Its performance is about 86% the performance of the native one on both cluster configurations (333.3 seconds versus 292.3 seconds and 491 seconds versus 422.3 seconds on local and cloud cluster respectively).

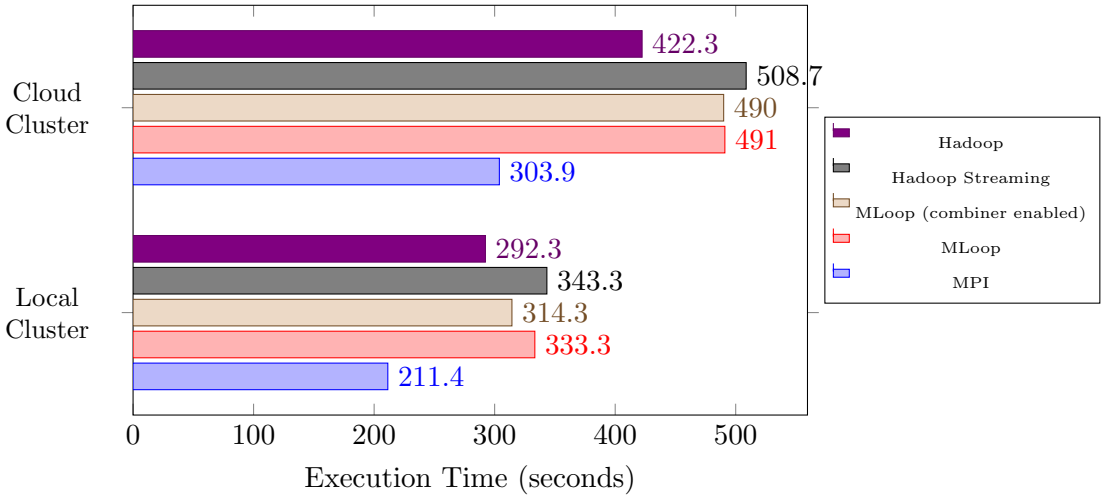

Figure 5.2: Performance on Graph problem

The result from Figure 5.2 also shows a remarkable point: the local cluster only needs around 2/3 the amount of time that the cloud cluster spends to solve the problem. This is a significant difference. This phenomenon is not hard to explain when remembering that this is a processor-intensive problem. The cloud cluster is able to run 8 truly parallel threads at the same time (8 machines * 1 cores * 1 thread/core) while the local cluster has 16 truly parallel threads (4 cores * 2 threads/core + 4 cores * 1 thread/core + 4 cores * 1 thread/core). Moreover, each machine in the local cluster is more powerful as we can see in the specification mentioned on the previous section. Therefore, solving the problem with MPI on local cluster achieves better result. Indeed, the result from this problem is also consistent with the result of Word Count problem when MPI also shows better performance on local cluster. The processing power of each machine in the cluster is also the explanation for the performance of Hadoop cluster in this experiment. The local cluster is able to process the task faster on each phase of the job (map and reduce phase) so that the overall performance achieved is better.

Figure 5.3 depicts the performance of cluster on 17-Queen problem. Because the operation in the reduce phase of the MapReduce algorithm does not really do any aggregation, the combiner will not help to optimize anything. Hence, in this experiment, MLoop with combiner enabled is not evaluated. As described before, this is the processor-intensive problem. Therefore, we can observe the familiar pattern in the result. The local cluster performs better on every parallel architecture, MPI and the family of Hadoop. The amount of time that MPI and the native Hadoop spend on local cluster is about as 60% as

on cloud cluster. On the other hand, MLoop and Hadoop Streaming on local cluster only saves about 20% execution time on cloud cluster. Among tested solutions, MPI continues showing its strengths in performance. The native Hadoop, the best in the Hadoop family, is slower about 32% and 23% on local and cloud cluster respectively.

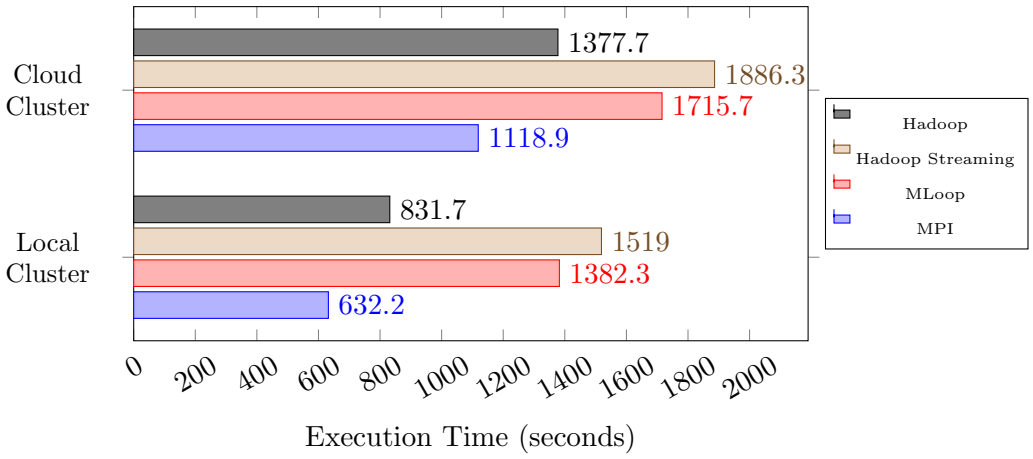

Figure 5.3: Performance on 17-Queen problem

One thing that we can observer from the result is that MLoop provides an improvement of about 9% in performance compared with Hadoop Streaming on both local and cloud cluster. It reduces the execution time from 1519 seconds to 1382.3 seconds and from 1886.3 seconds to 1715.7 seconds on local and cloud cluster respectively. This improvement saves a lot of time in long tasks. This experiment also confirms the promising of MLoop in comparison with the native Hadoop. MLoop achieves 1377.7/1715.7 = 80% the performance of the native Hadoop on cloud cluster. However, MLoop only obtains about 60% the native Hadoop performance on local cluster when it needs 1382.3 seconds to process while the native Hadoop only spends 831.7 seconds.

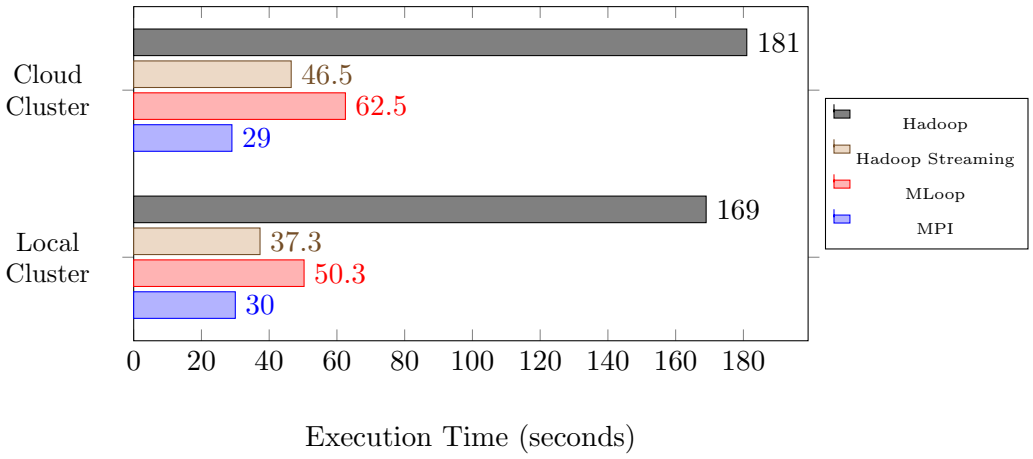

Figure 5.4: Performance on Summation problem

The result of the last experiment is shown in Figure 5.4. In this experiment, the native Hadoop performs really bad in comparison with others. Its execution time is from 3.4 to

4 times the execution time of the worst one in remaining solutions. This comes from the nature of the programming language. With MPI, Summation algorithm is implemented in C++ using GMP library for arbitrary precision arithmetic. With MLoop and Hadoop Streaming for Standard ML, this algorithm is implemented in SML. However, SML also uses GMP as its multiple precision arithmetic library. GMP library offers very fast operations for huge operands compared with the implementation in Java which is used in the native Hadoop. Therefore, it is reasonable that MPI, MLoop and Hadoop Streaming for SML perform very well in comparison with the native Hadoop. Moreover, MPI is still the best option when completing the task in only about 30 seconds. MLoop also shows very good result with 50 seconds. Hadoop Streaming for SML works better than MLoop this time with 37.3 and 46.5 seconds on local and cloud cluster respectively. The operations which are executed by MLoop and Hadoop Streaming for SML are the same because they use same code. The only difference between MLoop and Hadoop Streaming for SML in this experiment is their communication with the MapReduce framework. Therefore, the difference in performance of MLoop and Hadoop Streaming comes from the communication cost. Hadoop Streaming uses system pipes while MLoop communicates with the MapReduce framework via socket. Therefore, Hadoop Streaming suffers from the cost of converting between the (key,value) pair and the line of bytes as input/output of pipes. Summation problem is a processor-intensive task with very few communication between processes/workers. Each process/worker only receives two inputs from the mapper, the starting and the ending number of the number sequence which is need to calculate the sum. When they finish, a reducer receives the partial sums from mappers and calculates the final result. Hence, the cost of converting in Hadoop Streaming is small. Furthermore, using socket is as not cost-effective as using pipe. Therefore, MLoop is slower than Hadoop Streaming in this case.

Scalability

In this metric, we conduct two tests: Word Count with different sizes of data and with variant sizes of cluster. The tests only provide reasonable results if the cluster is homogeneous so that we can eliminate the effect of unexpected factors. Therefore, these experiments are conducted on Google cloud cluster. However, we only have a limited resources with maximum cluster size of 8. As a result, the analysis of scalability with respect to resources is just appropriate for small clusters.

Input Size	Execution Time (s)/ Ratio per Unit			
	MPI	MLoop	Hadoop Streaming	Hadoop
2 GB (1x)	184.882/1	276/1	276.667/1	235.667/1
4 GB (2x)	445.782/1.21	749/1.36	820/1.48	590/1.25
6 GB (3x)	620.133/1.12	1154/1.39	1194.5/1.44	961.5/1.36
8 GB (4x)	892.111/1.21	1559/1.41	1628.5/1.47	1225/1.30

Table 5.4: Word Count with different sizes of data

We conduct the first test with various sizes of input data: 2 GB, 4 GB, 6 GB and 8 GB. The result is reported in Table 5.4. Figure 5.5 visualizes the ratio per unit of each solution at different sizes of input data: 2x (4GB), 3x (6 GB) and 4x (8 GB). It is expected that the value is greater than one and the smaller it is, the better scalability is. From this visualization, we can say that MPI provides the best scalability upon this dataset. When

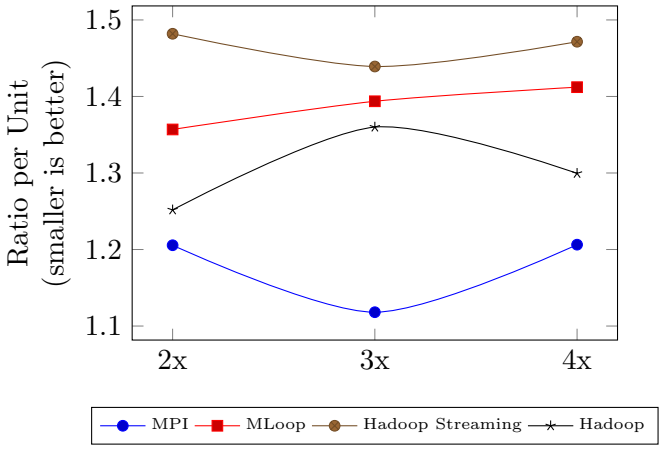

Figure 5.5: Scalability on Word Count with different input sizes

increasing the size of data 2 times, 3 times and 4 times, the execution time of MPI solution increase correspondingly with a nearly same rate, the ratio per unit which is between 1.1 and 1.2 is close to 1. The native Hadoop is the next candidate. It offers a ratio per unit ranged between 1.25 and 1.36, which is not too far from the ideal solution. MLoop is a bit worse than the native Hadoop when the ratio per unit is from 1.36 to 1.41. Hadoop Streaming shows that it has the least scalability when its ratio per unit is very close to 1.5 which means that the execution time almost increases 1.5 times faster than the amount of data does.

Cluster Size	Execution Time (s)/ Constant of Proportionality			
	MPI	MLoop	Hadoop Streaming	Hadoop
2 nodes (1x)	578.4085/1	1356.5/1	1441.5/1	1104/1
4 nodes (2x)	308.766/1.07	702/1.04	729.5/1.01	619.5/1.12
6 nodes (3x)	229.925/1.19	464/1.03	480/1.00	369.5/1.00
8 nodes (4x)	195.502/1.35	292/0.86	315/0.87	237.5/0.86

Table 5.5: Word Count with different cluster sizes

In the second test, Word Count problem with input data 2 GB is solved with different cluster sizes ranged from 2 nodes to 8 nodes. Table 5.5 tells the detailed result. The constant of proportionality a is illustrated in Figure 5.6. The red line is the value a in the ideal case. It is expected that all the values are greater than the value of the ideal case which is one. MPI provides very good scalability when double the size of cluster, $a = 1.07 \approx 1$. However, when the cluster size continues increasing, a also increases, from 1.07 to 1.19 and then 1.35. This means that MPI is less efficient with bigger cluster size, up to 8 nodes. This result is reasonable because increasing the number of nodes in the cluster also increases communication cost.

On the contrary, the Hadoop family achieves a very good value of a which is very close to 1 or even below 1. It is extraordinary that the Hadoop family has a less than 1 ($a \approx 0.86$) in 8 node cluster. It means that the execution time even decreases faster than the cluster size increases. This is impossible if the cluster utilizes as much of its strength as possible at any time. Consider the native Hadoop cluster. When changing the cluster size from 2 nodes to 4 nodes (2 times), from 4 nodes to 6 nodes (1.5 times) and 6

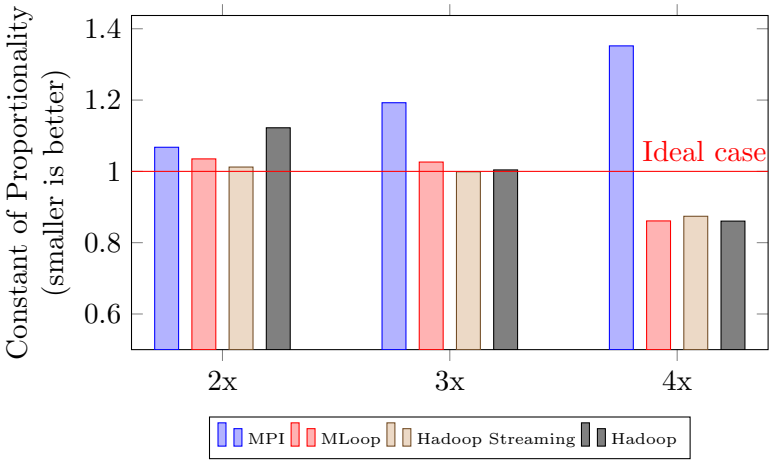

Figure 5.6: Scalability on Word Count with different cluster sizes

nodes to 8 nodes (1.33 times), the execution time reduces 1.78, 1.67 times and 1.55 times respectively. This pattern shows that Hadoop cluster achieves better performance with bigger cluster sizes. However, because there is not enough resources to test bigger cluster size than 8 nodes, we do not have not enough data to correctly analyze this pattern. With this test, we just know that Hadoop works well with cluster size of 6 or 8 than 2 or 4. The inefficiency of very small cluster (2 or 4 nodes) makes Hadoop performs too badly and spends too much time to complete. As a result, there is a big improvement when we compare the performance between small cluster and big cluster. This is the explanation for above extraordinary result.

Fault Tolerance

This feature is easy to test. One of the nodes is shut down. If the program is still able to complete, the current solution provides good fault tolerance. Otherwise, it does not. We should remember that MPI is a specification, not an implementation. Therefore, in this criteria, we mention fault tolerance as property of an MPI program coupled with an MPI implementation.

The actual result confirms that MPI (OpenMPI 1.8.1) does not provide fault tolerance ability by default. When one of nodes fails, the entire program also fails. Although OpenMPI supports BLCR checkpoint/restart system [2], it requires configuration and not transparent to users. On the other hand, Hadoop supports fault tolerance by default as it is designed for that. When a node fails, all the tasks running on the failed node will be re-scheduled on another node of the cluster. Actual result shows that the Word Count on Hadoop finishes after about 15 minutes in case of one node fails. In normal case, Hadoop solves this problem in about 5 minutes. This is because by default Hadoop considers a node as failed if it has sent no heartbeat during the period of ten minutes. This interval can be configured by users. However, when we shutdown the master node, the Hadoop job then fails to complete. This is a serious issue. However, in Hadoop cluster, this failure is unlikely because the chance of a particular failing machine is low compared with the chance of failing an arbitrary machine in cluster. Furthermore, Hadoop provides mechanisms to recover from this failure. Therefore, it is not a big matter.

In summary, fault tolerance in MPI is supported poorly. Hadoop, on the other hand, provides well-designed, fault tolerant services. Hadoop cluster is robust with failures.

Development Effort

The development effort of parallel programs could be evaluated in many aspects. They may include the amount of time to design, write program; how readable the code is; the effort to maintain or debug program. These aspects are hard to evaluated properly: the amount of time to develop programs depends on the experience of developer and the complexity of algorithms that need to be implemented while two other aspects are hard to quantify. Therefore, in the scope of this evaluation, we use two simple features to represent the effort of develop programs. First, the number of code lines can be used to demonstrate the complexity of algorithms. Therefore, it makes sense to use that number to measure the development effort. Second, instead of measuring the effort to maintain the program quantitatively, we consider it as three levels of difficulty: easy, medium and hard.

Problem	The number of code lines			
	MPI	MLoop	Hadoop Streaming	Hadoop
Word Count	316	27	41	96
Graph Search	433	83	109	244
17-Queens	257	104	135	191
Summation	91	33	59	90

Table 5.6: The number of code lines

The results shown in Table 5.6 confirm that writing program in Standard ML provides the most compact code. That is because Standard ML is the functional programming language. Writing programs in SML means describing what the solution is rather than how to solve it. Therefore, the program written in SM is shorter than in other non-functional programming languages. Indeed, MLoop and Hadoop Streaming provides shorter programs as seen in the result. The reason that Hadoop Streaming program is longer than corresponding program in MLoop is that there are operations which need to take place to parse standard input into (key,value) input pair in Hadoop Streaming program with SML.

From previous experiments, we know that MPI provides the best performance. However, in return, the programs in MPI is much more complicated compared with programs written in Hadoop family. With complicated algorithms such as Graph Search, 17-Queens and even with simple program like Word Count, program written in MPI is 3 to 10 times as long as in MLoop. That is because developers must handle different aspects which in Hadoop they don't have to do such as reading, distributing data, managing parallel architecture.

The native Hadoop also provides compact codes. Programs written by Java MapReduce is twice or three times as long as programs written in MLoop.

The level of difficulty to develop parallel programs is evaluated by my experience when developing different algorithms with these parallel solutions. The thought is that parallel models in Hadoop is also easy to develop because of its clear and simple ideas. I can write programs simple and easy to understand, easy to debug and maintain without a lot of effort. Therefore, the native Hadoop (Java MapReduce), MLoop and Hadoop Streaming for SML have the difficulty level "easy". On the contrary, I spent a lot of time to write programs with MPI. Most of time is spent to develop the parallel architecture, the way

to split the task, the data as well as manage the synchronization between processes. The effort which is needed to analyze the bug or discover the bug is also a major problem. Finally, the complexity in the language is also a concern. Writing MPI in C++ requires a big care about working with pointers and memory management. These tasks are simplified in languages such as Java, Standard ML, but not in C/C++. Hence, the difficulty level of writing programs in MPI is considered as "hard".

5.3 Discussion

Before MLoop is introduced, Hadoop Streaming is the first solution which is considered to develop distributed and parallel programs with Standard ML. Results from experiments show that performance of Hadoop Streaming is not so far from the native Hadoop. In most of our experiments, Hadoop Streaming achieves about 73% to 85% performance of the native Hadoop, except for the experiment on local cluster with 17-Queens problem where Hadoop Streaming only acquires about 55% the native Hadoop's performance. Hadoop Streaming only provides better performance in Summation problem when SML's strength on operations for huge operands is utilized. From [17], we also know that Hadoop Streaming jobs which are programmed with C++ is slower than the native Hadoop in data-intensive jobs, but not in processor-intensive jobs. Therefore, we may conclude that Hadoop Streaming (with any language) suffers from performance losses in data-intensive jobs. Its performance in processor-intensive jobs depends on the programming language which is used in Hadoop Streaming.

On the other hand, although Hadoop Streaming (with SML) inherits fault tolerance ability of Hadoop, it does not keep original scalability as analysis in the section 5.2.2. The most advantage of using Hadoop Streaming is that MapReduce programs can be written in Standard ML. This helps to reduce the development effort because developers are allowed to use their favorite programming language. Nevertheless, developers can only define map, reduce and combiner functions for MapReduce job running on Hadoop cluster. We cannot control other aspects of Hadoop framework such as the input reader, output writer, the counter, etc. Hadoop Streaming also does not support any mechanisms to chain multiple MapReduce jobs. This limits the application of Hadoop Streaming on problems which needs iterative jobs to resolve.

The results from experiments confirm that MLoop is a good replacement for Hadoop Streaming to develop parallel programs with SML. In most metrics, MLoop beats Hadoop Streaming when providing better features. Except for Summation problem, MLoop provides a 7% to 10% performance improvement in most cases. As a result, MLoop can achieve about 80% - 95% performance of the native Hadoop in most cases. In the worst case - 17-Queens on local cluster, MLoop gets 60% the native Hadoop's performance. MLoop is only left behind by Hadoop Streaming in problems which only require very few data communications like Summation. MLoop is also better in terms of scalability and development effort. Moreover, MLoop provides developers more control of on operating MapReduce jobs. It allows developers to control the way Hadoop framework parses input splits and writes output (key, value) pairs.

Good performance that MLoop brings back proves the our approach is right. Actually, our approach is inspired by the work in [23] where Pydoop showed a big improvement compared with Streaming (Python scripts). However, in terms of performance, the current approach is not the best choice. The approach used in Perldoop [10] brings back much more

improvements on the performance of Streaming (Perl scripts): at least five times faster. As mentioned in section 1.1.3, this approach converts the Hadoop Streaming scripts in Perl into Java codes and therefore allows to run every part of job inside Hadoop framework.

MLoop still has several weaknesses. Results from experiments force us to reconsider the efficiency of combiner implementation of MLoop. It is expected that combiner helps to reduce the execution time of MapReduce jobs. But combiner only helps to reduce a small amount of time in Graph Search problem on local cluster while it almost does not provide any improvement on cloud cluster. Using combiner in MLoop even makes the performance worse as we saw in Word Count problem. This shows that the current implementation is not efficient in communications between MLoop and Hadoop Pipes when calling *combiner* function. Although MLoop achieves better scalability in comparison with Hadoop Streaming, it is still worse than the native Hadoop. Furthermore, like Hadoop Streaming, Hadoop cannot take advantage of Hadoop counter as well as freely access to HDFS. We also cannot chain MapReduce jobs in MLoop.

If the performance of the applications is the key feature that we need, the native Hadoop and MPI are the best candidates. Choosing them means that developers have to change their programming languages. They have to use Java with the native Hadoop or C/C++ with MPI. From experimental results, MPI provides best performance. MPI only spends about 70% - 82% execution time of the native Hadoop. In the best case, it only uses half amount of execution time. This result is consistent to the results reported in [17]. MPI also provides better scalability when it can exploit cluster's resources more efficiently. But this scalability decreases when increasing the cluster's size as it also increases communication costs. This requires developers to maintain a more complicated parallel model to communicate more efficiently between processes. Furthermore, MPI is really poor in fault tolerant ability if compared with Hadoop. The effort that developers need to spend to develop MPI programs is also much more than to write MapReduce programs. Take Word Count program as an example. In MapReduce, we only need to define two simple steps: tokenize the tokens in *map* phase and aggregate the result in *reduce* phase. In contrast, developers have to write their own codes to distribute the data, manage communication between processes, gather the result, etc. with MPI (section 3.3.1). In short, MPI puts lots of burden to developers. This is worth to consider because in many applications like try and error analysis, we need an approach which is quick, easy to develop rather than spending too much time just for implementing the algorithms. In these cases, MPI is not a wise choice. Hadoop and MLoop are much better solutions. Therefore, if we care about the robustness and simpleness of our programs, the native Hadooop and MLoop are the promising choices.

One question may arise when we see the performance of MPI in comparison with others is that why we do not provide a Standard ML API for MPI but instead for Hadoop if MPI provides so good performance. This is because MPI has to suffer from many weaknesses as mentioned above in order to provide that performance. These weaknesses are very hard to overcome. Meanwhile, MLoop is slower in performance but we can easily solve it with larger cluster.

Another worth highlighting point that we could draw from experiments is the performance of cluster on different configurations. We have two configurations: a smaller cluster of stronger nodes (local cluster) and a bigger cluster of weaker nodes (cloud cluster). We can recognize that the local cluster performs better on processor-intensive problems such as 17-Queens, Graph Search and Summation. On Graph Search problem, the local cluster

only spends two thirds the execution time of cloud cluster to complete the task. The results allows us to confirm that the processing power of nodes in cluster is an important factor which determines the performance on processor-intensive problems. On the other hand, we could see that Hadoop cluster performs better on cloud cluster with Word Count (a data-intensive problem). In this case, cloud cluster which has bigger available memory for YARN containers provides higher degree of parallelization. Therefore, it is easy to understand the result. However, MLoop is an exception for this judgment. As analyzed before, the overhead of communications of MLoop when increasing the degree of parallelization makes it worse on cloud cluster. Then we come to a conclusion: with data-intensive problems, MLoop prefers a smaller cluster of stronger nodes to big cluster of low-end commodity nodes. Finally, we realize from the experimental results that MPI always performs better on cluster of stronger machines. MPI which requires developers to manage the parallel model of program by themselves can utilize cluster's processing-resources efficiently and in an optimal way for a particular problem.

Above comments may make us want to come back to the "scale-up" approach to build cluster: purchasing high-end machines to build small clusters. Then we will have efficient clusters with high performance. This may be true. However, the main challenge here is that it is not cost effective since the costs of high-end machines do not scale linearly. A machine with twice as many processors is often more than twice as expensive. In 2009, Barroso and Hölzle [20] conducted a TPC-C benchmark [3] to compare the cost-efficiency between cluster based on a high-end server and one based on a low-end server. The results showed a difference in cost-efficiency of over a factor of four in favor of the low-end server. Thus, the "scale out" is preferred to "scale up". That is what MapReduce model aims to. To achieve a desired performance, it is recommended to increase the cluster's size rather than increase the processing power of machines. However, we can use a hybrid approach to reduce the disadvantage of "scale up". Instead of using high-end machines to build a small cluster, we can have a bigger cluster but with mid-end commodity machines.

5.4 Guidelines

Experiences drawn from experiments can be summarized into more general guide-lines which are worth to consider when choosing the distributed and parallel solution for actual problem. The first guide-line is to define the ultimate goal of the desired solution. We may want our solution to have good performance, high scalability, easy to develop, etc. Understanding this clearly is the first step to find the best appropriate solution. We could not find out a thing that ourselves do not know about it. In particularly, if we want to have a solution that can provide performance as good as possible, MPI is the answer. Otherwise, if we also require the solution to have other good features such as robustness, easy to develop and maintain, Hadoop family provides promising choices.

The strengths of each solution for a particular problem is the next thing that we need to care. Each parallel approach has its own strengths and weaknesses depending on the application. The native Hadoop is the most reasonable choice for data-intensive problems. With processor-intensive problems, MPI, Hadoop Streaming (C++ scripts) and Hadoop Pipes may be are recommended because of the efficiency of C++ language. However, it is impossible to have a common formula for every problem. For example, we could say that in general the native Hadoop is the best one among Hadoop, Hadoop Streaming, MLoop. However, in some situation, it is not true any more. The Summation problem gives us an

example. The strength of Standard ML with operations on very big operands is exploited in this experiment. Therefore, MLoop and Hadoop Streaming (SML scripts) is 3 or 4 times faster than the native Hadoop.

There are other aspects that we need to look at to decide the best suitable approach for our problem. A detailed analysis on all of them is unnecessary. Instead, we should focus on several critical aspects which can influence most to the operation of the solution. They are the aspects which lots of researchers focus on such as scalability, development effort, fault tolerance, operating and maintenance cost, energy cost.

The forth guide-line is that we may develop our own solution to meet our requirements. Promising results offered by MLoop are the motivation for doing that. Although MLoop still suffers from lots of drawbacks, it shows that it is not possible to provide an acceptable solution through extending Hadoop. In this way, we still take advantage of great features of Hadoop while also making use of good aspects of our favorite programming languages. Hence, if we cannot change programming language but want to have the performance beyond Hadoop Streaming, just implement an extension of Hadoop Pipes. Architecture and necessary technologies are already discussed in section 4.3.

The last guide-line that we want to mention is about the cluster's configuration. Although "scale out" is preferred to "scale up" approach, we should think about the hybrid approach. Instead of using a big cluster of low-end machines, we can use a slightly smaller cluster of mid-end machines. This can increase the cost a bit, but the performance might be improved a lot. Therefore, we recommend a cluster of more powerful machines. Besides that, the configuring a cluster also depends on the type of cluster. If we want to build a MPI cluster, the more total processing cores is, the better performance is. If we want to build a Hadoop cluster, we should provide more available memory in order to increase the degree of parallelization.

Chapter 6

Conclusions and Future Work

This chapter sums up the work and concludes this thesis. Section 6.1 gives a summary of what have been done and the contribution of this work. Section 6.2 suggests possible future work to improve the current solution as well as recommendations for future research.

6.1 Summary

Big Data has become a fact of the world. It is rapidly becoming one of the driving forces behind the global economy. It leads to a strong demand of efficient processing and storage models. This thesis aims to provide a distributed and parallel approach for programs in Standard ML. Instead of inventing everything from scratch, we extend Hadoop Pipes framework to make it support MapReduce programs defined in SML. This thesis responses to two research objectives:

1. Develop a Standard ML API for Hadoop which allows to write MapReduce programs in SML that can run on Hadoop cluster.

2. Evaluate new API (library) to show its performance in comparison with existing large-scale solutions.

3. Provide some guide-lines to find out the best suitable solution for actual issue.

The library that we develop is called MLoop. It allows developers to use Standard ML to define MapReduce programs. In current version of MLoop, developers can define the *map*, *reduce* and *combiner* functions of MapReduce jobs. It also allows developers to customize the way of parsing input splits into input (key, value) pairs to feed into *map* function. Moreoever, the outputs of *reduce* function can be modified before actually writing to Hadoop Distributed File System. MLoop inherits great features from Hadoop such as fault tolerance, scalability, easy programming model, free system-level detail while it stills keep the strengths of Standard ML.

Experimental results confirm that MLoop provides better performance than existing approach provided by Hadoop, Hadoop Streaming. In general, MLoop performs worse than the native Hadoop. However, in most experiments, it achieves about 80% to 95% performance of the native Hadoop in terms of execution time. This is a very promising result. In some special case, MLoop even beats the native Hadoop in terms of execution time. Summation problem is an example. In this case, Standard ML is much faster than

Java in processing operations with big numbers. As a result, Summation program in MLoop requires much less amount of time than in Hadoop Java.

However, the results also show that the combiner of MLoop is not efficient and does not work as expected. It does not provide significant improvement in cases which it should. The combiner even reduces the performance on Word Count problem. This reduction comes from the communication overhead of using combiner in MLoop. Furthermore, the approach of developing MLoop makes it suffer from inevitable shortcomings. MLoop cannot take advantage of Hadoop counter. This prevents developers from approaching some useful programming patterns with Hadoop. MLoop also does not provide any mechanisms to chain multiple MapReduce jobs to solve complicated problems. This makes it difficult or even impossible to implement a class of problems which needs iterative MapReduce tasks to solve.

Experiences from experiments are summarized and generalized into guide-lines. These guide-lines together with MLoop are the main contribution of this thesis. The guide-lines remind developers to consider aspects to find out the most suitable approach for their actual issues with big data or large-scale computations.

6.2 Future Work

The current version of MLoop suffers from several drawbacks. Some of them are inevitable while others can be improved. The first thing to improve MLoop is to reduce the communication overhead of combiner in MLoop. The intermediate output pairs of map function are sent back to Hadoop Pipes framework and buffered there before sending back to combiner function of MLoop. Therefore, a suggestion is that doing all of those stuffs in MLoop. In that way, we can save a lot of communication costs between MLoop and Pipes framework.

In current version, developers indirectly works with HDFS through Record Reader and Writer. This limits the operations to several pre-defined actions. Providing HDFS API in MLoop which allows developers to read,write files, get information of files, directories is a big improvement. This provides developers more control in file operations. Then several business logics can be implemented more easily.

Hadoop framework allows developers to customize the partitioner functions. They specify the reducer to which an intermediate key-value pair belongs. Furthermore, Hadoop provides application-level counters that can be updated and retrieved by developers. Supporting partitioner and counter in MLoop are also considered in future.

Because of limitation on available resources, our experiment were conducted with small clusters. It is interesting to know the performance of different large-scale parallel solutions on bigger clusters. Therefore, one of our future work after improving MLoop is to evaluate it with other solutions on bigger clusters. At that time, we can get the results from clusters which have sizes closer to real cases. As a result, we can have better analyses.

Finally, promising result of this thesis motivates us to go further to provide a complete solution for Standard ML to write programs which can run on Hadoop cluster. Remember that YARN supports different kinds of jobs not only MapReduce. Therefore, one suggestion for future research is to study how to develop application running on YARN. Then we can build a YARN application based on MapReduce model but for Standard ML. This requires a lot of effort. However, we will be able to achieve a really good framework for Standard ML which can utilize full features of MapReduce model.

Bibliography

[1] Automatic Design of Algorithms Through Evolution. http://www-ia.hiof.no/~rolando/. [Online; accessed 09-May-2015].

[2] Berkeley Lab Checkpoint/Restart (BLCR) for LINUX. http://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/. [Online; accessed 09-May-2015].

[3] Transaction Processing Performance Council. http://www.tpc.org/. [Online; accessed 09-May-2015].

[4] MLton's Foreign Function Interface. http://mlton.org/ForeignFunctionInterface, 2014. [Online; accessed 09-May-2015].

[5] "Online in 60 seconds – A Year Later". http://blog.qmee.com/online-in-60-seconds-infographic-a-year-later/, July 2014. [Online; accessed 09-May-2015].

[6] Folding@home. https://folding.stanford.edu/, 2015. [Online; accessed 09-May-2015].

[7] RHadoop Wiki. https://github.com/RevolutionAnalytics/RHadoop/wiki, Feb 2015. [Online; accessed 09-May-2015].

[8] SETI@home. http://setiathome.ssl.berkeley.edu/, 2015. [Online; accessed 09-May-2015].

[9] Swig-3.0 documentation. http://www.swig.org/Doc3.0/SWIGDocumentation.html, 2015. [Online; accessed 09-May-2015].

[10] José M Abuín, Juan C Pichel, Tomás F Pena, Pablo Gamallo, and Marcos Garcia. Perldoop: Efficient execution of perl scripts on hadoop clusters. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 766–771. IEEE, 2014.

[11] RajashekharM. Arasanal and DaanishU. Rumani. Improving mapreduce performance through complexity and performance based data placement in heterogeneous hadoop clusters. In Chittaranjan Hota and PradipK. Srimani, editors, *Distributed Computing and Internet Technology*, volume 7753 of *Lecture Notes in Computer Science*, pages 115–125. Springer Berlin Heidelberg, 2013.

[12] Michele Banko and Eric Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th Annual Meeting on Association*

for Computational Linguistics, ACL '01, pages 26–33, Stroudsburg, PA, USA, 2001. Association for Computational Linguistics.

[13] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In Bernhard Schölkopf, John C. Platt, and Thomas Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.

[14] CouchBase. Dealing with Memcached Challenges. `http://info.couchbase.com/rs/northscale/images/Couchbase_WP_Dealing_with_Memcached_Challenges.pdf`, 2012. [Online; accessed 09-May-2015].

[15] Jeffrey Dean and Sanjay Ghemawat. *MapReduce : Simplified Data Processing on Large Clusters*, volume 51, pages 1–13. ACM, 2004.

[16] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[17] Mengwei Ding, Long Zheng, Yanchao Lu, Li Li, Song Guo, and Minyi Guo. More convenient more overhead: The performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 307–313, New York, NY, USA, 2011. ACM.

[18] Zhenhua Guo and Geoffrey Fox. Improving mapreduce performance in heterogeneous network environments and resource utilization. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 714–716, Washington, DC, USA, 2012. IEEE Computer Society.

[19] Apache Hadoop. HDFS Federation. `http://hadoop.apache.org/docs/r2.4.1/hadoop-project-dist/hadoop-hdfs/Federation.html`, 2015. [Online; accessed 09-May-2015].

[20] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[21] John and Cailin. Breadth-first graph search using an iterative map-reduce algorithm. `http://www.johnandcailin.com/blog/cailin/breadth-first-graph-search-using-iterative-map-reduce-algorithm`, 2009. [Online; accessed 09-May-2015].

[22] Sharanjit Kaur, Rakhi Saxena, Dhriti Khanna, and Vasudha Bhatnagar. Comparing data processing frameworks for scalable clustering. In *The Twenty-Seventh International Flairs Conference*, 2014.

[23] Simone Leo and Gianluigi Zanetti. Pydoop: A python mapreduce and hdfs api for hadoop. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 819–825, New York, NY, USA, 2010. ACM.

[24] Jure Leskovec. Web data: Amazon movie reviews. `https://snap.stanford.edu/data/web-Movies.html`, April 2015. [Online; accessed 09-May-2015].

[25] Jimmy Lin. *Data-intensive text processing with MapReduce*. Morgan, Claypool, San Rafael, 2010.

[26] Peter Pacheco. *An Introduction To Parallel Programming*, chapter Distributed-Memory Programming with MPI, pages 83 – 140. Morgan Kaufmann, 2011.

[27] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[28] Vignesh Prajapati. *Big data analytics with R and Hadoop*. Packt Publishing Ltd, 2013.

[29] Sangwon Seo, Ingook Jang, Kyungchang Woo, Inkyo Kim, Jin-Soo Kim, and Seungryoul Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, Aug 2009.

[30] Daisy Tang. Lecture Notes: B-Trees. `https://www.cpp.edu/~ftang/courses/CS241/notes/b-tree.htm`, 2015. [Online; accessed 09-May-2015].

[31] Qun Liao Yulu Yang Tao Li Tao Gu, Chuang Zuo. Improving mapreduce performance by data prefetching in heterogeneous or shared environments. *International Journal of Grid and Distributed Computing*, 6, 2013.

[32] Tom White. *Hadoop: The Definitive Guide, 3rd Edition*, chapter Meet Hadoop, page 8. O'Reilly, 2012.

[33] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010.

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

Appendix A

Working with MLoop

In this appendix, we cover how to use our library MLoop to build MapReduce programs in Standard ML on Hadoop cluster.

A.1 Prerequisites

MLoop is written in C++ and Standard ML. MLton is used to compile Standard ML in MLoop. So you need to install a MLton on your machine. Besides, MLoop executables run on Hadoop cluster. Hence, you also need to install Java, version 6 or later.

A.2 Installation

A.2.1 Installing Hadoop

We cover here main step to configure a Hadoop cluster in Ubuntu. You can find a full version here `http://n0where.net/multi-node-hadoop-cluster/`.

Adding dedicated Hadoop system user

We will use a dedicated Hadoop user account for running Hadoop. While that's not required but it is recommended, because it helps to separate the Hadoop installation from other software applications and user accounts running on the same machine.

```
$ sudo addgroup Hadoop
$ sudo adduser -ingroup Hadoop hduser
```

We created a user named `hduser` and assigned him to group `Hadoop`.

Configuring SSH access

Because Hadoop works with a lot of nodes, it will need some techniques to perform its operations. Indeed, it uses pass-phrase-less SSH for this purpose. Generating a public/private key pair and placing it locally on every node in the cluster is the simplest way. At that point, the master node can provide the private key and access to slave nodes.

To generate a SSH key for `hduser` account, we need:

1. Login as `hduser` with `sudo`

2. Run this command to generate the key: `$ ssh-keyegen -t rsa -P ""`
   It will ask to provide the file name in which to save the key, press "Enter" so that it will generate the key at '/home/hduser/.ssh'

To enable SSH access to your local machine with this newly created key.
`$ cat $HOME/.ssh/id_rsa.pub >> $HOME/.ssh/authorized_keys`

Disabling IPv6

Ubuntu is using 0.0.0.0 IP for different Hadoop configurations. Hence, we need to disable IPv6. At first,

`$ sudo gedit /etc/sysctl.conf`

Then add the following lines to the end of the file:

```
#disable ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

Installing Hadoop

Run following command to download Hadoop version 2.2.0, unpack it and change the owner.

```
$ wget https://archive.apache.org/dist/hadoop/core/hadoop-2.2.0/hadoop
    -2.2.0.tar.gz
$ tar -xvzf hadoop-2.2.0.tar.gz
$ sudo chown -R hduser:Hadoop hadoop-2.2.0
```

Add the following lines at end of .bashrc file

```
export HADOOP_HOME=/home/hduser/hadoop-2.2.0
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

Add JAVA_HOME to libexec/hadoop-config.sh at beginning of the file

```
$ vi /home/hduser/hadoop-2.2.0/libexec/hadoop-config.sh
...
export JAVA_HOME='/usr/local/Java/jdk1.7.0_45'
...
```

Add JAVA_HOME to hadoop/hadoop-env.sh at beginning of the file

```
$ vi /home/hduser/hadoop-2.2.0/etc/hadoop/hadoop-env.sh
...
export JAVA_HOME='/usr/local/Java/jdk1.7.0_45'
...
```

Check Hadoop installation

```
$ cd /home/hduser/hadoop-2.2.0/bin
$ ./hadoop version
Hadoop 2.2.0
...
```

At this point Hadoop installed in your node.
Create folder `tmp`

```
$ mkdir -p $HADOOP_HOME/tmp
```

Configuring multi-node cluster

Add IP address of Master and all Slaves to /etc/hosts – for both master and all the slave nodes.

Password-less ssh from host `master` to host `slave1`:

```
$ ssh-copy-id -i /home/hduser/.ssh/id_dsa.pub hduser@slave
$ ssh slave1
```

Only at Master node, add all slaves into `slaves` file.

```
$ vi /home/hduser/hadoop-2.2.0/etc/hadoop/slaves
slave1
slave2
...
```

Add the properties in following hadoop configuration file which is availabile under `$HADOOP_CONF_DIR`.

core-site.xml

```
<property>
  <name>fs.default.name</name>
  <value>hdfs://master:9000</value>
</property>
<property>
  <name>hadoop.tmp.dir</name>
  <value>/home/hduser/hadoop-2.2.0/tmp</value>
</property>
```

hdfs-site.xml

```
<property>
  <name>dfs.replication</name>
  <value>2</value>
</property>
<property>
  <name>dfs.permissions</name>
  <value>false</value>
</property>
```

Note: Here, replication values is 2 [one master and one slave]. If you have more slaves put replication value based on that. But it is recommend that replication value should be 3 for small cluster.

mapred-site.xml

```
<property>
  <name>mapreduce.framework.name</name>
  <value>yarn</value>
</property>
```

yarn-site.xml

```
<property>
  <name>yarn.nodemanager.aux-services</name>
  <value>mapreduce.shuffle</value>
```

```
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.resource-tracker.address</name>
    <value>master:8025</value>
  </property>
  <property>
    <name>yarn.resourcemanager.scheduler.address</name>
    <value>master:8030</value>
  </property>
  <property>
    <name>yarn.resourcemanager.address</name>
    <value>master:8040</value>
  </property>
```

Only at Master node, format the namenode. This is just for the first time of setting up cluster.

```
/home/hduser/hadoop-2.2.0/bin$ ./hadoop namenode -format
```

Starting Hadoop

We only need to start the process at Master node. The processes at slave nodes will automatically start.

```
/home/hduser/hadoop-2.2.0/sbin$ ./start-dfs.sh
/home/hduser/hadoop-2.2.0/sbin$ ./start-yarn.sh
```

We use `start-dfs.sh` to start namenode and datanode and `start-yarn.sh` to start resourcemanager and nodemanager.

A.2.2 Installing MLoop

At first, we need to install MLton

`$ sudo apt-get install mlton-compiler`

Install some required packages to compile MLoop.

```
$ sudo apt-get install uuid-dev
$ sudo apt-get install libssl-dev
```

Download and extract MLoop package from `https://github.com/mywish07/mloop/archive/master.zip`. This zip file contains MLoop library in folder *mloop-master*. MLoop library is also provided in the CD, at folder *source code/MLoop*.

```
$ wget https://github.com/mywish07/mloop/archive/master.zip
$ unzip master.zip
```

This package contains some example programs written in MLoop such as Word Count (wordcount.sml), Graph Search (graph.sml), N-Queen (queen.sml and queen2.sml) and Summation (sum.sml).

To compile wordcount.sml into executable program in Hadoop, use following `build` utility. This assumes that you already set environment variable `$HADOOP_HOME`.

```
$ cd mloop-master
$ ./build main.sml
```

In the same folder, the output executable file is `mloop`.
To execute this file in Hadoop cluster,

1. Start Hadoop
2. Copy this file into HDFS.

```
$ cd /home/hduser/hadoop-2.2.0/bin
$ ./hdfs dfs -copyFromLocal /home/hduser/mloop-master/mloop /
```

3. Use following command to start a job in Hadoop cluster

```
$ ./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.
    java.recordwriter=true -input <input> -output <output folder> -program /
    mloop -reduces 3
```

   This command will execute the job defined by file `mloop` in the folder `/` of HDFS. Before in this case, we do not use Record Reader and Writer of MLoop, the command has two options with values `true`. In case of using Record Reader, we set `hadoop.pipes.java.recordreader=false`.

   Option `-reduces` is optional. This is used to set the number of the reducers. Here we use three reducers.

A.3 MLoop Documentation

MLoop is a project which aims to create a library for Standard ML to run in parallel with Hadoop. MLoop provides a simple way to write parallel programs in Standard ML with the Map and Reduce concepts of Hadoop. Running in the Hadoop cluster, parallel programs written with MLoop inherit great features of Hadoop: scalability and fault tolerance.

MLoop is developed as an extension of Hadoop Pipes which delegates tasks to SML functions. The SML executable file interacts with Hadoop Framework to process the task. Hadoop Framework initializes the job, divides it into smaller tasks which can be run in parallel. Each task will be handled by the SML functions.

A.3.1 MLoop Structures

MLoop provides several structures to interact between SML functions and the Hadoop Pipes framework. These structures play an important role in the library.

MapContext

The MapContext structure provides two utility functions for manipulating the map task: storing the address of MLoopMapper C++ class, an implementation of Mapper C++ abstract class; emitting the intermediate key-value pairs to the Hadoop Pipes framework.
Interface

val **setAddress**: MLton.Pointer.t ->unit
val **emit**: string * string ->unit

Description

setAddress adress
: store the address of MLoopMapper C++ class. This method is reserved for the internal use. The developer has nothing to deal with it.

emit (key, value)
: emit the intermediate key-value pair to the Hadoop Pipes framework.

ReduceContext

The ReduceContext structure provides utility functions for manipulating the reduce task: retrieve the value associated with a specific key; emit final key-value pair.
Interface

val **setAddress**: MLton.Pointer.t ->unit
val **emit**: string * string ->unit
val **getInputValue** : unit ->string
val **nextValue**: unit ->bool
val **getValueSet**: unit ->string list

Description

setAddress adress
: store the address of MLoopReducer C++ class. This method is reserved for the internal use. The developer has nothing to deal with it.

emit (key,value)
: emit the output (key, value) pair to the Hadoop Pipes framework.

getInputValue ()
: return the next value associated with current key.

nextValue ()
: check whether or not the next value associated with current key exists.

getValueSet ()
: return all the values associated with current key. Developer must be careful. The result list may consumes a lot of memory in case there are too many values.

Reader

The Reader structure provides utility functions for working with custom Record Reader.
Interface

val **getOffset**: unit ->Int64.int
val **getBytes_read**: unit ->Int64.int
val **updateOffset_bytesConsumed**: Int64.int * Int64.int ->unit
val **getOffsetNow**: unit ->Int64.int
val **get**: unit ->Reader
val **seekHdfs**: Int64.int ->unit

Description

getOffset ()
: return the offset of the current input split which corresponds to this map task.

getBytes_read ()
: return the number of bytes consumed by the reader for the last read operation.

updateOffset_bytesConsumed (offset, bytesConsumed)
: update the current offset of the reader in the file and the number of bytes consumed for the last read operation.

getOffsetNow ()
: return the current offset of the reader in the file.

get ()
: return the reader.

seekHdfs offset
: seek to given offset in file.

Writer

The Writer structure provides utility functions for working with custom Record Writer.
Interface

val **emit**: string * string ->unit

Description

emit (key,value)
: write the key-value pair to the output file in HDFS.

A.4 Running Sample Programs in Hadoop

Use above steps to build executables for programs in MLoop and rename it. For example, assume mloop-wc, mloop-graph, mloop-sum, mloop-queen and mloop-queen2 are the executables of wordcount.sml, graph.sml, sum.sml, queen.sml and queen2.sml, respectively. We also upload all of these executables into root directory of HDFS. This is not required, you can upload them into any folder in HDFS.

A.4.1 Word Count

Run following commands to run Word Count programs in Hadoop cluster with n reducers.

```
$ cd /home/hduser/hadoop-2.2.0/bin
$ ./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.
    java.recordwriter=true -input <input> -output <output folder> -program /
    mloop-wc -reduces n
```

A.4.2 Summation

Run following commands to run Summation programs in Hadoop cluster.

```
$ cd /home/hduser/hadoop-2.2.0/bin
$ ./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.
    java.recordwriter=true -input <input> -output <output folder> -program /
    mloop-sum -reduces 1
```

A.4.3 17-Queens

In 17-Queens, we need to run in sequence different jobs to find the complete solution. At first, we run five jobs defined in mloop-queen.sml, then we run job defined in mloop-queens2.sml. Because MLoop does not support chaining jobs, we have to do it manually. Create a bash file with following content:

```
#!/usr/bin/env bash
./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.java.
    recordwriter=true -input /input-nqueen -output /output-mloop/queen1 -
    program /mloop-queen -reduces 8
i=1
for j in 2 3 4 5
do
  ./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.
      java.recordwriter=true -input /output-mloop/queen$((i++)) -output /
      output-mloop/queen$j -program /mloop-queen -reduces 8
done

./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.java.
    recordwriter=true -input /output-mloop/queen5 -output /output-mloop/
    queen -program /mloop-queen2 -reduces 8
```

Then execute the bash file to run the jobs.

A.4.4 Graph Search

We have the same situation with previous problem. Therefore, we also create a bash file with following content.

```
#!/usr/bin/env bash
./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.java.
    recordwriter=true -input /input-graph -output /output-mloop/graph1 -
    program /mloop-graph -reduces 6
i=1
for j in 2 3 4 5
do
  ./mapred pipes -D hadoop.pipes.java.recordreader=true -D hadoop.pipes.
      java.recordwriter=true -input /output-mloop/graph$((i++)) -output /
      output-mloop/graph$j -program /mloop-graph -reduces 6
done
```

Execute the bash file to run the jobs.

Appendix B

Program Source Code

In this appendix, we include the full program source code for MapReduce programs.

B.1 Hadoop Java Programs

B.1.1 Word Count

Listing B.1: WordCount.java

```
1  import org.apache.hadoop.conf.Configuration;
2  import org.apache.hadoop.fs.FileSystem;
3  import org.apache.hadoop.fs.Path;
4  import org.apache.hadoop.io.IntWritable;
5  import org.apache.hadoop.io.Text;
6  import org.apache.hadoop.mapreduce.Job;
7  import org.apache.hadoop.mapreduce.Mapper;
8  import org.apache.hadoop.mapreduce.Reducer;
9  import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
10 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
12 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
13
14 import java.io.IOException;
15 import java.util.StringTokenizer;
16
17 public class WordCount {
18     public static void main(String[] args) throws IOException,
19             InterruptedException, ClassNotFoundException {
20         long startTime = System.currentTimeMillis();
21         Path inputPath = new Path(args[0]);
22         Path outputDir = new Path(args[1]);
23
24         Configuration conf = new Configuration(true);
25         Job job = Job.getInstance(conf, "wordcount");
26         job.setJarByClass(WordCount.class);
27
28         job.setMapperClass(WordCountMapper.class);
29         job.setReducerClass(WordCountReducer.class);
30         if (args.length > 2)
31             job.setNumReduceTasks(Integer.parseInt(args[2]));
32         else
33             job.setNumReduceTasks(4);
34
35         // Specify key / value
36         job.setOutputKeyClass(Text.class);
37         job.setOutputValueClass(IntWritable.class);
38
39         // Input
40         FileInputFormat.addInputPath(job, inputPath);
```

```
41          job.setInputFormatClass(TextInputFormat.class);
42
43          // Output
44          FileOutputFormat.setOutputPath(job, outputDir);
45          job.setOutputFormatClass(TextOutputFormat.class);
46
47          // Delete output if exists
48          FileSystem hdfs = FileSystem.get(conf);
49          if (hdfs.exists(outputDir))
50              hdfs.delete(outputDir, true);
51
52          // Execute job
53          int code = job.waitForCompletion(true) ? 0 : 1;
54          long endTime = System.currentTimeMillis();
55          System.out.println("Total time: " + (endTime - startTime) + "ms");
56          System.exit(code);
57
58      }
59
60      public static class WordCountMapper extends
61              Mapper<Object, Text, Text, IntWritable> {
62          private final IntWritable ONE = new IntWritable(1);
63          private Text word = new Text();
64
65          public WordCountMapper() {
66          }
67          public void map(Object key, Text value, Context context)
68                  throws IOException, InterruptedException {
69              StringTokenizer itr = new StringTokenizer(value.toString());
70              while (itr.hasMoreTokens()) {
71                  word.set(itr.nextToken());
72                  context.write(word, ONE);
73              }
74          }
75      }
76
77      public static class WordCountReducer extends
78              Reducer<Text, IntWritable, Text, IntWritable> {
79          public void reduce(Text text, Iterable<IntWritable> values,
80                             Context context) throws IOException, InterruptedException {
81              int sum = 0;
82              for (IntWritable value : values) {
83                  sum += value.get();
84              }
85              context.write(text, new IntWritable(sum));
86          }
87      }
88
89  }
```

B.1.2 Graph Search

Listing B.2: GraphSearch.java

```
1  import org.apache.commons.logging.Log;
2  import org.apache.commons.logging.LogFactory;
3  import org.apache.hadoop.conf.Configuration;
4  import org.apache.hadoop.conf.Configured;
5  import org.apache.hadoop.fs.Path;
6  import org.apache.hadoop.io.IntWritable;
7  import org.apache.hadoop.io.LongWritable;
8  import org.apache.hadoop.io.Text;
9  import org.apache.hadoop.mapreduce.Counters;
10 import org.apache.hadoop.mapreduce.Job;
11 import org.apache.hadoop.mapreduce.Mapper;
12 import org.apache.hadoop.mapreduce.Reducer;
13 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
14 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
15 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
16 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
17 import org.apache.hadoop.util.Tool;
```

```
18 import org.apache.hadoop.util.ToolRunner;
19
20 import java.io.IOException;
21 import java.util.ArrayList;
22 import java.util.List;
23
24 public class GraphSearch extends Configured implements Tool {
25     public static final Log LOG = LogFactory.getLog(GraphSearch.class);
26     static int printUsage() {
27         System.out.println("graphsearch [-m <num mappers>] [-r <num reducers>]");
28         ToolRunner.printGenericCommandUsage(System.out);
29         return -1;
30     }
31
32     public static void main(String[] args) throws Exception {
33         int res = ToolRunner.run(new Configuration(), new GraphSearch(), args);
34         System.exit(res);
35     }
36
37     public int run(String[] args) throws Exception {
38         int iterationCount = 0;
39         long terminateValue = 1;
40         while (terminateValue > 0) {
41             String input;
42             if (iterationCount == 0)
43                 input = "/input-graph";
44             else
45                 input = "/output/output-graph-" + iterationCount;
46             String output = "/output/output-graph-" + (iterationCount + 1);
47
48             Job job = getJobConf(args);
49             FileInputFormat.setInputPaths(job, new Path(input));
50             FileOutputFormat.setOutputPath(job, new Path(output));
51
52             job.waitForCompletion(true);
53             Counters jobCnt = job.getCounters();
54             terminateValue = jobCnt.findCounter(counter_enum.ITERATION).getValue();
55             iterationCount++;
56         }
57         return 0;
58     }
59
60     private Job getJobConf(String[] args) throws IOException {
61         Configuration conf = getConf();
62         Job job = Job.getInstance(conf, "graphsearch");
63         job.setJarByClass(GraphSearch.class);
64
65         job.setInputFormatClass(TextInputFormat.class);
66         job.setOutputFormatClass(TextOutputFormat.class);
67         job.setOutputKeyClass(Text.class);
68         job.setOutputValueClass(Text.class);
69
70         job.setMapperClass(MapClass.class);
71         job.setReducerClass(Reduce.class);
72
73         for (int i = 0; i < args.length; ++i) {
74             if ("-r".equals(args[i])) {
75                 job.setNumReduceTasks(Integer.parseInt(args[++i]));
76             }
77         }
78         return job;
79     }
80
81     public static enum counter_enum {ITERATION}
82
83     public static class MapClass extends Mapper
84             <LongWritable, Text, Text, Text> {
85         protected void map(LongWritable key, Text value, Context context) throws IOException,
                InterruptedException {
86             Node node = new Node(value.toString());
87             // For each GRAY node, emit each of the edges as a new node (also GRAY)
```

```
88              if (node.getColor() == Node.Color.GRAY) {
89                  String edges = node.getEdges();
90                  if (edges != null && !"NULL".equals(edges)){
91                      for (String v : edges.split(",")) {
92                          context.write(new Text(v), new Text("NULL|" + (node.getDistance() + 1) +
                                "|GRAY"));
93                      }
94                  }
95                  // We're done with this node now, color it BLACK
96                  node.setColor(Node.Color.BLACK);
97              }
98
99              // If the node came into this method GRAY, it will be output as BLACK
100             context.write(new Text(node.getId()), node.getLine());
101         }
102     }
103
104     public static class Reduce extends Reducer
105             <Text, Text, Text, Text> {
106         protected void reduce(Text key, Iterable<Text> values,
107                               Context context) throws IOException, InterruptedException {
108             String edges = "NULL";
109             int distance = Integer.MAX_VALUE;
110             Node.Color color = Node.Color.WHITE;
111
112             for (Text value : values) {
113                 Node u = new Node();
114                 u.updateInfo(value.toString());
115                 if (!"NULL".equals(u.getEdges())) {
116                     edges = u.getEdges();
117                 }
118
119                 // Save the minimum distance
120                 if (u.getDistance() < distance) {
121                     distance = u.getDistance();
122                 }
123
124                 // Save the darkest color
125                 if (u.getColor().ordinal() > color.ordinal()) {
126                     color = u.getColor();
127                 }
128             }
129
130             Node n = new Node();
131             n.setDistance(distance);
132             n.setEdges(edges);
133             n.setColor(color);
134             context.write(key, new Text(n.getLine()));
135             if (color == Node.Color.GRAY)
136                 context.getCounter(counter_enum.ITERATION).increment(1L);
137         }
138     }
139 }
```

B.1.3 N-Queens

In this program, first five job will use the map function which is defined by MapClass class. These job will do breadth-first search to find all possible ways of putting first five queens on the board. The last job will use the map function which is defined by MapClass2 class. This job finds all the solutions for each given board with five queens placed.

Listing B.3: NQueens.java

```
1 import org.apache.hadoop.conf.Configuration;
2 import org.apache.hadoop.conf.Configured;
3 import org.apache.hadoop.fs.Path;
4 import org.apache.hadoop.io.IntWritable;
5 import org.apache.hadoop.io.LongWritable;
6 import org.apache.hadoop.io.Text;
7 import org.apache.hadoop.mapreduce.Counters;
```

```
 8 import org.apache.hadoop.mapreduce.Job;
 9 import org.apache.hadoop.mapreduce.Mapper;
10 import org.apache.hadoop.mapreduce.Reducer;
11 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
12 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
15 import org.apache.hadoop.util.Tool;
16 import org.apache.hadoop.util.ToolRunner;
17
18 import java.io.IOException;
19 import java.util.Random;
20
21 public class NQueens extends Configured implements Tool {
22     static int n = 17;
23     public static void main(String[] args) throws Exception {
24         if (args.length > 0) {
25             n = Integer.parseInt(args[0]);
26             System.out.println(n);
27         }
28         int res = ToolRunner.run(new Configuration(), new NQueens(), args);
29         System.exit(res);
30     }
31
32     @Override
33     public int run(String[] args) throws Exception {
34         int iterationCount = 0;
35
36         String input;
37         String output;
38         while (iterationCount < 5) {
39             if (iterationCount == 0) {
40                 input = "/input-nqueen";
41             } else {
42                 input = "/output/output-nqueen-" + iterationCount;
43             }
44
45             output = "/output/output-nqueen-" + (iterationCount + 1);
46             Job job = getJobConf(1, args);
47             FileInputFormat.setInputPaths(job, new Path(input));
48             FileOutputFormat.setOutputPath(job, new Path(output));
49
50             job.waitForCompletion(true);
51             Counters jobCnt = job.getCounters();
52             iterationCount++;
53         }
54         input = "/output/output-nqueen-" + iterationCount;
55         output = "/output/output-nqueen-" + (iterationCount + 1);
56         Job job = getJobConf(2, args);
57         FileInputFormat.setInputPaths(job, new Path(input));
58         FileOutputFormat.setOutputPath(job, new Path(output));
59         job.waitForCompletion(true);
60         return 0;
61     }
62
63     private Job getJobConf(int n, String[] args) throws IOException {
64         Configuration conf = getConf();
65         Job job = Job.getInstance(conf, "nQueens");
66         job.setJarByClass(NQueens.class);
67
68         job.setInputFormatClass(TextInputFormat.class);
69         job.setOutputFormatClass(TextOutputFormat.class);
70         job.setOutputKeyClass(IntWritable.class);
71         job.setOutputValueClass(Text.class);
72
73         if (n == 1)
74             job.setMapperClass(MapClass.class);
75         else
76             job.setMapperClass(MapClass2.class);
77         job.setReducerClass(Reduce.class);
78         if (args.length > 0)
```

```
79              job.setNumReduceTasks(Integer.parseInt(args[0]));
80          else
81              job.setNumReduceTasks(6);
82
83          return job;
84      }
85
86      public static enum counter_enum {ITERATION}
87
88      public static class MapClass extends Mapper<LongWritable, Text, IntWritable, Text> {
89          public MapClass() {
90          }
91
92          @Override
93          protected void map(LongWritable key, Text value, Context context) throws IOException,
                InterruptedException {
94              String val = value.toString().trim();
95              if ("START".equals(val)) {
96                  for (int i = 0; i < n; i++) {
97                      context.write(new IntWritable(1), new Text(String.valueOf(i)));
98                  }
99                  return;
100             }
101
102             String[] map = val.split("-");
103             int[] board = new int[map.length];
104             for (int i = 0; i < map.length; i++) {
105                 board[i] = Integer.parseInt(map[i]);
106             }
107             Random r = new Random();
108             // Put new queen on new column
109             for (int row = 0; row < n; row++) {
110                 if (safe(board, row)) {
111                     context.write(new IntWritable(r.nextInt(10)), new Text(val + "-" + row));
112                 }
113             }
114         }
115
116         private boolean safe(int[] board, int row) {
117             int currentColumn = board.length;
118             for (int i = 1; i <= currentColumn; i++) {
119                 int preRow = board[currentColumn - i];
120                 if (preRow == row || preRow == row - i || preRow == row + i)
121                     return false;
122             }
123             return true;
124         }
125     }
126
127     public static class MapClass2 extends Mapper<LongWritable, Text, IntWritable, Text> {
128         public MapClass2() {
129         }
130
131         @Override
132         protected void map(LongWritable key, Text value, Context context) throws IOException,
                InterruptedException {
133             String val = value.toString().trim();
134             String[] map = val.split("-");
135             int[] board = new int[n];
136             int len = map.length;
137             for (int i = 0; i < len; i++) {
138                 board[i] = Integer.parseInt(map[i]);
139             }
140
141             Random r = new Random();
142             // Currently, board is filled with first 'len' columns
143             // Find the solution for this branch
144             int column = len; // column 'len' + 1
145             board[column] = -1;
146             while (column >= len) {
147                 int row = -1;
```

```
148                 do {
149                     row = board[column];
150                     board[column] = row = row + 1;
151                 } while (row < n && !safe(board, column));
152                 if (row < n) {
153                     if (column < n - 1) {
154                         board[++column] = -1;
155                     } else { // found the board
156                         String s = String.valueOf(board[0]);
157                         for (int i = 1; i < n; i++) {
158                             s += "-" + board[i];
159                         }
160                         context.write(new IntWritable(r.nextInt(10)), new Text(s));
161                     }
162                 } else {
163                     column--;
164                 }
165             }
166         }
167 
168         private boolean safe(int[] board, int column) {
169             int row = board[column];
170             for (int i = 1; i <= column; i++) {
171                 int preRow = board[column - i];
172                 if (preRow == row || preRow == row - i || preRow == row + i)
173                     return false;
174             }
175             return true;
176         }
177     }
178 
179     public static class Reduce extends Reducer<IntWritable, Text, Text, Text> {
180         @Override
181         protected void reduce(IntWritable key, Iterable<Text> values, Context context) throws
                IOException, InterruptedException {
182             for (Text val : values) {
183                 context.write(new Text(""), val);
184             }
185         }
186     }
187 }
```

B.1.4 Summation

Listing B.4: Sum.java

```
 1 import hadoop.in.action.template.MapClass;
 2 import hadoop.in.action.template.Reduce;
 3 import org.apache.hadoop.conf.Configuration;
 4 import org.apache.hadoop.conf.Configured;
 5 import org.apache.hadoop.fs.FSDataOutputStream;
 6 import org.apache.hadoop.fs.FileSystem;
 7 import org.apache.hadoop.fs.Path;
 8 import org.apache.hadoop.io.LongWritable;
 9 import org.apache.hadoop.io.Text;
10 import org.apache.hadoop.mapreduce.Job;
11 import org.apache.hadoop.mapreduce.Mapper;
12 import org.apache.hadoop.mapreduce.Reducer;
13 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
14 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
15 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
16 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
17 import org.apache.hadoop.util.Tool;
18 import org.apache.hadoop.util.ToolRunner;
19 
20 import java.io.IOException;
21 import java.math.BigInteger;
22 
23 public class Sum extends Configured implements Tool {
```

```
24      @Override
25      public int run(String[] args) throws Exception {
26          Configuration conf = getConf();
27          Job job = Job.getInstance(conf,"sum");
28          job.setJarByClass(Sum.class);
29
30          Path in = new Path(args[0]);
31          Path out = new Path(args[1]);
32          FileInputFormat.setInputPaths(job, in);
33          FileOutputFormat.setOutputPath(job, out);
34
35          job.setInputFormatClass(TextInputFormat.class);
36          job.setOutputFormatClass(TextOutputFormat.class);
37          job.setOutputKeyClass(Text.class);
38          job.setMapOutputValueClass(BigIntegerWritable.class);
39          job.setOutputValueClass(Text.class);
40
41          FileSystem hdfs = FileSystem.get(conf);
42          hdfs.setWriteChecksum(false);
43          for (int i = 0; i < 20; i++){
44              Path file = new Path(in, "part_" + i);
45              FSDataOutputStream outStream = hdfs.create(file);
46              BigInteger base = new BigInteger("500000000");
47              String start = base.multiply(BigInteger.valueOf((long)i)).add(BigInteger.ONE).
                    toString();
48              String end = base.multiply(BigInteger.valueOf((long) (i + 1))).toString();
49              outStream.writeUTF(start + " " + end);
50              outStream.close();
51          }
52
53          job.setMapperClass(SumMap.class);
54          job.setReducerClass(SumReduce.class);
55          int code = job.waitForCompletion(true) ? 0 : 1;
56          System.exit(code);
57          return 0;
58      }
59
60      public static void main(String[] args) throws Exception {
61          int res = ToolRunner.run(new Configuration(), new Sum(), args);
62          System.exit(res);
63      }
64
65      private static class SumMap extends Mapper<LongWritable, Text, Text, BigIntegerWritable>{
66          Text word = new Text("");
67          @Override
68          protected void map(LongWritable key, Text value, Context context) throws IOException,
                InterruptedException {
69              String val = value.toString().trim();
70              if ("".equals(val))
71                  return;
72              String[] temp = val.split(" ");
73
74              BigInteger start = new BigInteger(temp[0].trim());
75              BigInteger end = new BigInteger(temp[1].trim());
76              BigInteger sum = BigInteger.ZERO;
77              while(start.compareTo(end) <= 0){
78                  sum = sum.add(start);
79                  start = start.add(BigInteger.ONE);
80              }
81              context.write(word, new BigIntegerWritable(sum));
82          }
83      }
84
85      private static class SumReduce extends Reducer<Text, BigIntegerWritable, Text, Text>{
86          @Override
87          protected void reduce(Text key, Iterable<BigIntegerWritable> values, Context context)
                throws IOException, InterruptedException {
88              BigInteger sum = BigInteger.ZERO;
89              for (BigIntegerWritable value : values){
90                  sum = sum.add(value.getValue());
91              }
```

```
92              context.write(new Text("Total sum: "), new Text(sum.toString()));
93          }
94      }
95  }
```

B.2 MLoop Programs

B.2.1 Word Count

Listing B.5: wordcount.sml

```
1
2  fun mloop_map (key:string, value:string) =
3      let
4          val splitter = String.tokens(fn c => Char.isSpace c)
5          val words = splitter value
6
7      in
8          map (fn word => MapContext.emit (word,"1")) words;
9          ()
10     end
11
12 fun mloop_reduce (key:string)=
13     let
14         fun fromString str = let
15             val x = Int.fromString str
16             in
17                 Option.getOpt(x,0)
18             end
19         val sum = ref 0
20     in
21         while (ReduceContext.nextValue()) do
22             sum := (!sum) + (fromString (ReduceContext.getInputValue()));
23         ReduceContext.emit(key, Int.toString(!sum))
24     end
25
26 fun mloop_combine (key:string) =
27     mloop_reduce (key)
28
29 val useCombiner = true
```

B.2.2 Graph Search

Listing B.6: graph.sml

```
1  fun split (str,c) = String.tokens (fn ch => ch = c) str
2
3  fun tokens str = String.tokens (fn c => Char.isSpace c) str
4
5  fun join (l,separator) = let
6    fun j ([],sep,result) = result
7      | j ([h],sep,result) = result ^ h
8      | j (h::t, sep, result) = j (t,sep,result ^ h ^ sep)
9    in
10   j(l,separator,"")
11 end
12
13 val maxInt = Option.getOpt(Int.maxInt,1000000000)
14
15 (* info = EDGES|DISTANCE_FROM_SOURCE|COLOR| *)
16 fun unpackInfo info = let
17   val tmp = split (info,#"|")
18   val valid2 = if List.length tmp = 3 then true else raise Fail("Invalid info.")
19   val e::d::c::_ = tmp
20 in
21    (e,d,c)
22 end
23
24 (* str = ID    EDGES|DISTANCE_FROM_SOURCE|COLOR| *)
```

```
25 fun unpack str = let
26   val tmp = tokens str
27   val valid = if List.length tmp = 2 then true else raise Fail("Invalid key-value.")
28   val k::v::_ = tmp
29 in
30   (k, unpackInfo v)
31 end
32
33 fun pack {id=k:string,edges=e:string list,distance=d:int,color=c:string} = (join (e,",")) ^ "|" ^
       (Int.toString d) ^ "|" ^ c
34 fun newNode (identity,e, dist,c) = {id=identity,edges=e,distance=dist,color=c}
35
36 fun printNode (id, edges, dist, color) = MapContext.emit (id, edges ^ "|" ^ dist ^ "|" ^ color)
37 fun reduceNode (id, edges, dist, color) = ReduceContext.emit (id, edges ^ "|" ^ dist ^ "|" ^
     color)
38
39 fun mloop_map (key,value) = let
40   val (id,(edges,dist,color)) = unpack value
41   fun increaseDist dist = case (Int.fromString dist) of SOME t => Int.toString(t+1) | NONE => "
       Integer.MAX_VALUE"
42   fun parseEdges e = if e = "NULL" then [] else split (e,#",")
43  in
44   if color = "GRAY" then (
45   map (fn k => printNode (k,"NULL",(increaseDist dist),color)) (parseEdges edges);
46   printNode (id, edges,dist,"BLACK")
47   )
48    else
49   printNode (id,edges,dist,color)
50 end
51
52 fun mloop_reduce (key:string) = let
53   val dist = ref "Integer.MAX_VALUE"
54   val colour = ref "WHITE"
55   val edge = ref "NULL"
56   fun colorInt color = case color of "WHITE" => 0 | "GRAY" => 1 | "BLACK" => 2 | _ => ~1
57   fun maxColor (c1, c2) = let
58     val v1 = colorInt c1
59     val v2 = colorInt c2
60     in
61     if v1 < v2 then c2 else c1
62   end
63   fun minDist (d1, d2) = let
64   val v1 = case (Int.fromString d1) of SOME t => t | NONE => maxInt
65   val v2 = case (Int.fromString d2) of SOME t => t | NONE => maxInt
66   in
67   if v1 < v2 then d1 else d2
68   end
69   fun update (edges:string,distance:string,color:string) = (if not (edges = "NULL") then edge:=
       edges else ();
70            dist:= minDist (!dist, distance);
71            colour := maxColor(!colour,color))
72  in
73     while (ReduceContext.nextValue()) do
74         update (unpackInfo (ReduceContext.getInputValue()));
75     reduceNode (key,!edge,!dist,!colour)
76 end
```

B.2.3 N-Queens

Listing B.7: queen.sml

```
1 fun threat (x,y) (x',y') = let
2   val distance = x' - x
3   in
4     y = y' orelse y = y' - distance orelse y = y' + distance
5 end
6
7 fun conflict pos = List.exists (threat pos)
8
9 fun randInt ()= let
10   val t = Time.now()
```

```
  val b = Time.toMilliseconds t
  val bucket = b mod 10
in
  IntInf.toString bucket
end

fun codeBoard ([],str) = str
  | codeBoard ((_,row)::t, "") = codeBoard(t, Int.toString row)
  | codeBoard ((_,row)::t, str) = codeBoard(t, (Int.toString row) ^ "-" ^ str)

fun addQueen (qs,n) = let
  val col = 1 + List.length qs
  fun put row = if row > n then ()
  else if conflict (col,row) qs then put (row + 1)
  else (MapContext.emit(randInt (), codeBoard((col,row)::qs, "")); put (row + 1))
 in
     put 1
 end

fun append (row,nil) = [(1,row)]
  | append (row, (col,x)::t) = (col + 1, row) :: (col,x)::t

fun parseBoard value = let
  val splitter = String.tokens(fn c => c = #"-")
  fun toInt s = Option.getOpt (Int.fromString s, 0)
  val values = map toInt (splitter value)
  in foldl append nil values
end

fun mloop_map (key,value) = if value = "START" then addQueen ([],17)
  else addQueen(parseBoard value,17)

fun mloop_reduce (key:string)=
    while (ReduceContext.nextValue()) do
        ReduceContext.emit("", ReduceContext.getInputValue())
```

Listing B.8: queen2.sml

```
fun threat (x,y) (x',y') = let
  val distance = x' -  x
  in
    y = y' orelse y = y' - distance orelse y = y' + distance
end

fun conflict pos = List.exists (threat pos)

fun randInt ()= let
  val t = Time.now()
  val b = Time.toMilliseconds t
  val bucket = b mod 10
in
  IntInf.toString bucket
end

fun codeBoard ([],str) = str
  | codeBoard ((_,row)::t, "") = codeBoard(t, Int.toString row)
  | codeBoard ((_,row)::t, str) = codeBoard(t, (Int.toString row) ^ "-" ^ str)


fun fillQueen (qs,n,col,fc) = let
  fun put row = if row > n then fc ()
  else if conflict (col,row) qs then put (row + 1)
  else if col = n then (MapContext.emit(randInt (), codeBoard((col,row)::qs, "")); put (row + 1))
  else fillQueen((col,row)::qs,n,col+1, fn () => put (row+1))
 in
     put 1
 end

fun append (row,nil) = [(1,row)]
  | append (row, (col,x)::t) = (col + 1, row) :: (col,x)::t

```

```
34 fun parseBoard value = let
35   val splitter = String.tokens(fn c => c = #"-")
36   fun toInt s = Option.getOpt (Int.fromString s, 0)
37   val values = map toInt (splitter value)
38   in foldl append nil values
39 end
40
41 fun mloop_reduce (key:string)=
42     while (ReduceContext.nextValue()) do
43         ReduceContext.emit("", ReduceContext.getInputValue())
44
45
46 fun mloop_map (key,value) = let
47   val board = parseBoard value
48   val col = 1 + List.length board
49 in
50   fillQueen(board, 17, col, fn () => ())
51 end
```

B.2.4 Summation

Listing B.9: sum.sml

```
1
2 fun mloop_map (key:string, value:string) =
3     let
4         val splitter = String.tokens(fn c => Char.isSpace c)
5         val words = splitter value
6         fun toInt number = let
7             val x = IntInf.fromString number
8             in
9                 Option.getOpt(x,0)
10            end
11        fun sum (from:IntInf.int,to:IntInf.int, result:IntInf.int) = if from > to then result
12            else sum (from + 1, to, result + from);
13        val v = sum (toInt (List.nth(words,0)), toInt (List.nth(words,1)), 0);
14    in
15        MapContext.emit ("", IntInf.toString v)
16    end
17
18 fun mloop_reduce (key:string)=
19    let
20        fun fromString str = let
21            val x = IntInf.fromString str
22            in
23                Option.getOpt(x,0)
24            end
25        val sum = ref 0 : IntInf.int ref
26    in
27        while (ReduceContext.nextValue()) do
28            sum := (!sum) + (fromString (ReduceContext.getInputValue()));
29        ReduceContext.emit(key, IntInf.toString(!sum))
30    end
31
32 fun mloop_combine (key:string) =
33    mloop_reduce (key)
34
35 val useCombiner = true
```

B.3 MPI Programs

Here we only mention primary class for each problem. The full source code can be found in attached CD.

B.3.1 Word Count

Listing B.10: wordcount.cpp

```
#include "wordcount.h"
#include <iostream>
#include <fstream>
#include "mpi.h"
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>

using namespace std;

#define MIN(A,B) ((A) < (B)) ? (A) : (B)
#define BUFF_SIZE               256000000 // 256MB of data

int WordCount::count(int argc, char* argv[]) {
    typedef struct {
        int count;
        char word[30];
    } WordStruct;
    int blocks[2] = {1, 30};
    MPI_Datatype types[2] = {MPI_INT, MPI_CHAR};
    MPI_Aint displacements[2];
    MPI_Datatype obj_type;
    MPI_Aint charex, intex;

    int nTasks, rank;
    MPI::Init(argc, argv);
    nTasks = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();

    MPI_Type_extent(MPI_INT, &intex);
    MPI_Type_extent(MPI_CHAR, &charex);
    displacements[0] = static_cast<MPI_Aint> (0);
    displacements[1] = intex;
    MPI_Type_struct(2, blocks, displacements, types, &obj_type);
    MPI_Type_commit(&obj_type);

    // When creating out buffer, we use the new operator to avoid stack overflow.
    // If you are programming in C instead of C++, you'll want to use malloc.
    // Please note that this is not the most efficient way to use memory.
    // We are sacrificing memory now in order to gain more performance later.
    long long int *pLineStartIndex; // keep track of which lines start where. Mainly for
        debugging
    char *pszFileBuffer;
    int nTotalLines = 0;
    if (argc >= 3)
        buff_size = atoll(argv[2]);
    else
        buff_size = BUFF_SIZE;

    FILE* file;
    size_t total_size;
    if (rank == 0) {
        file = fopen("input.txt", "r+");
        struct stat filestatus;
        stat("input.txt", &filestatus);
        total_size = filestatus.st_size;
    }
    map<string, int> totalwordcount;
    while (1) {
        int cont = 1;
        if (rank == 0) {
            pLineStartIndex = new long long int[buff_size / 10]; // assume number of line is
                equal to number of character / 10
            pLineStartIndex[0] = 0;
            pszFileBuffer = readFile(file, total_size);
            if (pszFileBuffer == NULL)
                cont = 0;
        }
        nTotalLines = 0;
        MPI::COMM_WORLD.Bcast(&cont, 1, MPI::INT, 0);
```

```
69          if (cont == 0)
70              break;
71          if (rank == 0) {
72              char* ptr = strchr(pszFileBuffer, 10); // find a new line character
73              while (ptr != NULL) {
74                  nTotalLines++;
75                  pLineStartIndex[nTotalLines] = (ptr - pszFileBuffer + 1);
76                  ptr = strchr(ptr + 1, 10);
77              }
78              pLineStartIndex[nTotalLines + 1] = strlen(pszFileBuffer);
79          }
80
81          // Thread zero needs to distribute data to other threads.
82          // Because this is a relatively large amount of data, we SHOULD NOT send the entire
                dataset to all threads.
83          // Instead, it's best to intelligently break up the data, and only send relevant portions
                to each thread.
84          // Data communication is an expensive resource, and we have to minimize it at all costs.
85          char *buffer = NULL;
86          int totalChars = 0;
87          int portion = 0;
88          int startNum = 0;
89          int endNum = 0;
90
91
92          if (rank == 0) {
93              portion = nTotalLines / nTasks;
94              startNum = 0;
95              endNum = portion;
96              buffer = new char[pLineStartIndex[endNum] + 1];
97              strncpy(buffer, pszFileBuffer, pLineStartIndex[endNum]);
98              buffer[pLineStartIndex[endNum]] = '\0';
99              for (int i = 1; i < nTasks; i++) {
100                 // calculate the data for each thread.
101                 int curStartNum = i * portion;
102                 int curEndNum = (i + 1) * portion - 1;
103                 if (i == nTasks - 1) {
104                     curEndNum = nTotalLines;
105                 }
106                 if (curStartNum < 0) {
107                     curStartNum = 0;
108                 }
109
110                 // we need to send a thread the number of characters it will be receiving.
111                 int curLength = pLineStartIndex[curEndNum + 1] - pLineStartIndex[curStartNum];
112                 MPI_Send(&curLength, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
113                 if (curLength > 0)
114                     MPI_Send(pszFileBuffer + pLineStartIndex[curStartNum], curLength, MPI_CHAR, i
                            , 2, MPI_COMM_WORLD);
115             }
116             delete []pszFileBuffer;
117             delete []pLineStartIndex;
118         } else {
119             // We are not the thread that read the file.
120             // We need to receive data from whichever thread
121             MPI_Status status;
122             MPI_Recv(&totalChars, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
123             if (totalChars > 0) {
124                 buffer = new char[totalChars + 1];
125                 MPI_Recv(buffer, totalChars, MPI_CHAR, 0, 2, MPI_COMM_WORLD, &status);
126                 buffer[totalChars] = '\0';
127             }
128         }
129
130         // Do the search
131         map<string, int> wordcount;
132         int size = 0;
133         WordStruct* words = NULL;
134         if (buffer != NULL) {
135             char* word = strtok(buffer, " ,.\r\n");
136             while (word != NULL) {
```

```
137                 if (wordcount.find(word) == wordcount.end())
138                     wordcount[word] = 1;
139                 else
140                     wordcount[word]++;
141                 word = strtok(NULL, " ,.\r\n");
142             }
143             delete []buffer;
144 
145             size = wordcount.size();
146             if (size > 0) {
147                 words = new WordStruct[size];
148                 int i = 0;
149                 for (map<string, int>::iterator it = wordcount.begin(); it != wordcount.end(); it
                        ++) {
150 
151                     strcpy(words[i].word, (it->first).c_str());
152                     words[i].count = it->second;
153                     i++;
154                 }
155             }
156         }
157 
158         // At this point, all threads need to communicate their results to thread 0.
159         if (rank == 0) {
160             // The master thread will need to receive all computations from all other threads.
161             MPI_Status status;
162 
163             // We need to go and receive the data from all other threads.
164             // aggregate the result for current chunk of data
165             for (int i = 1; i < nTasks; i++) {
166                 int sz;
167                 MPI_Recv(&sz, 1, MPI_INT, i, 3, MPI_COMM_WORLD, &status);
168                 if (sz > 0) {
169                     WordStruct* local_words = new WordStruct[sz];
170                     MPI_Recv(local_words, sz, obj_type, i, 4, MPI_COMM_WORLD, &status);
171 
172                     for (int j = 0; j < sz; j++) {
173                         totalwordcount[local_words[j].word] += local_words[j].count;
174                     }
175                     delete []local_words;
176                 }
177             }
178 
179             for (map<string, int>::iterator it = wordcount.begin(); it != wordcount.end(); it++)
                    {
180                 totalwordcount[it->first] += it->second;
181             }
182         } else {
183             // We are finished with the results in this thread, and need to send the data to
                    thread 1.
184             // The destination is thread 0
185             MPI_Send(&size, 1, MPI_INT, 0, 3, MPI_COMM_WORLD);
186             if (size > 0) {
187                 MPI_Send(words, size, obj_type, 0, 4, MPI_COMM_WORLD);
188             }
189         }
190         wordcount.clear();
191         if (words != NULL && size > 0)
192             delete []words;
193     }
194     if (rank == 0) {
195         fclose(file);
196         // Display the final calculated value
197         ofstream out("output.txt");
198         for (map<string, int>::iterator it = totalwordcount.begin(); it != totalwordcount.end();
                it++) {
199             out << it->first << ": " << it->second << "\n";
200         }
201         totalwordcount.clear();
202         out.close();
203     }
```

```
204     MPI_Finalize();
205     return 0;
206 }
207
208 void WordCount::print(char* str, int start, int len) {
209     cout << "@";
210     for (int i = 0; i < len; i++) {
211         if (str[start + i] == '\r')
212             cout << "EOL";
213         else if (str[start + i] == '\n')
214             cout << "NL";
215         else
216         cout << str[start + i];
217     }
218     cout << "@" << endl;
219 }
220
221 char* WordCount::readFile(FILE* file, size_t fileSize) {
222     long long readsize = MIN(buff_size, fileSize - lastPosition);
223     if (readsize <= 0)
224         return NULL;
225     char* str = new char[readsize + 1];
226     fseek(file, lastPosition, SEEK_SET); // read file from last position
227     fread(str, 1, readsize, file);
228
229     // trim the end of string: remove part of string after the last space
230     long start = 0;
231     if (readsize > 50)
232         start = readsize - 50;
233     // check from last 50 characters
234     char* ptr = strchr(&str[start], ' ');
235
236     if (ptr == NULL || readsize < buff_size) { // return entire string
237         lastPosition = fileSize;
238         str[readsize] = '\0';
239         return str;
240     }
241     char* pre = NULL;
242     while (ptr != NULL) {
243         pre = ptr;
244         ptr = strchr(ptr + 1, ' ');
245     }
246
247     int rSize = pre - str + 1;
248     if (pre != NULL) {
249         *pre = '\0';
250     }
251
252     // update current read position
253     lastPosition += rSize;
254     return str;
255 }
```

B.3.2 Graph Search

The Graph Search problem is solved in class `BFS` in bfs.cpp.

Listing B.11: bfs.cpp

```
 1 #include <mpi.h>
 2 #include "bfs.hpp"
 3 #include "LineReader.hpp"
 4 #include <sys/stat.h>
 5 #include <stdio.h>
 6 #include <vector>
 7 #include <string.h>
 8 #include <string>
 9 #include <sstream>
10 #include <functional>
11
12 #include <fstream>
```

```
13 using namespace std;
14
15 #define LENG 0
16 #define DATA 1
17 #define RESPONSE 2
18
19 const int LOW = 1;
20 const int HIGH = 2;
21 const int NO_DATA = 3;
22 const int HAVE_DATA = 4;
23 const int MORE_DATA = 5;
24
25 vector<string> split(const char* strChar, char delimiter) {
26     vector<string> internal;
27     stringstream ss(strChar); // Turn the string into a stream.
28     string tok;
29
30     if (delimiter == ' ' || delimiter == '\t') {
31         while (ss) {
32             ss >> tok;
33             internal.push_back(tok);
34         }
35     } else {
36         while (getline(ss, tok, delimiter)) {
37             internal.push_back(tok);
38         }
39     }
40     return internal;
41 }
42
43 int hashKey(string data, int n) {
44     hash<string> str_hash;
45     size_t value = str_hash(data);
46     return value % n;
47 }
48
49 void storeNewNode(int size, string key, string data, vector<string>* &otherdata) {
50     int id = hashKey(key, size);
51     otherdata[id].push_back(data);
52 }
53
54 vector<string> parseNode(const char* str) {
55     vector<string> result;
56     vector<string> internal = split(str, ' ');
57     result.push_back(internal[0]);
58     vector<string> edges = split(internal[1].c_str(), '|');
59     result.insert(result.end(), edges.begin(), edges.end());
60     return result;
61 }
62
63 void createNode(string& result, string id, string nodes, string dist, string color) {
64     result.clear();
65     result.append(id);
66     result.append("\t");
67     result.append(nodes);
68     result.append("|");
69     result.append(dist);
70     result.append("|");
71     result.append(color);
72     result.append("|");
73 }
74
75 void increaseDist(string& output, string distance) {
76     if (distance.compare("Integer.MAX_VALUE") == 0) {
77         output = distance;
78         return;
79     }
80
81     int value = stoi(distance);
82     output = to_string(value + 1);
83 }
```

```
84
85 void minDist(string& output, string dist1, string dist2) {
86     if (dist1 == "Integer.MAX_VALUE") {
87         output = dist2;
88         return;
89     } else if (dist2 == "Integer.MAX_VALUE") {
90         output = dist1;
91         return;
92     }
93
94     int value1 = stoi(dist1);
95     int value2 = stoi(dist2);
96     output = value1 < value2 ? dist1 : dist2;
97 }
98
99 int encodeColor(string color) {
100     if (color == "WHITE")
101         return 0;
102     if (color == "GRAY")
103         return 1;
104     if (color == "BLACK")
105         return 2;
106     return -1;
107 }
108
109 void maxColor(string& output, string color1, string color2) {
110     int c1 = encodeColor(color1);
111     int c2 = encodeColor(color2);
112     output = c1 < c2 ? color2 : color1;
113 }
114
115 void swap(string& s1, string& s2) {
116     string temp;
117     temp = s1;
118     s1 = s2;
119     s2 = temp;
120 }
121
122 void siftDown(vector<string>& data, int start, int n) {
123     while (start * 2 + 1 < n) {
124         // children are 2*i + 1 and 2*i + 2
125         int child = start * 2 + 1;
126         // get bigger child
127         if (child + 1 < n && data[child] < data[child + 1]) child++;
128
129         if (data[start] < data[child]) {
130             swap(data[start], data[child]);
131             start = child;
132         } else
133             return;
134     }
135 }
136
137 void sort(vector<string> &data) { // heap-sort
138     int size = data.size();
139     if (size == 0)
140         return;
141     for (int k = size / 2; k >= 0; k--) {
142         siftDown(data, k, size);
143     }
144
145     while (size > 1) {
146         swap(data[size - 1], data[0]);
147         siftDown(data, 0, size - 1);
148         size--;
149     }
150 }
151
152 void sendData(int target, vector<string> &data, int count) {
153     MPI_Send(&count, 1, MPI_INT, target, LENG, MPI_COMM_WORLD);
154     for (int i = 0; i < count; i++) {
```

```
155         int data_len = data[i].length();
156         MPI_Send(&data_len, 1, MPI_INT, target, LENG, MPI_COMM_WORLD);
157         MPI_Send(data[i].c_str(), data_len, MPI_CHAR, target, DATA, MPI_COMM_WORLD);
158     }
159 }
160
161 void receive_resp(int from, vector<string> &output) {
162     int len;
163     MPI_Recv(&len, 1, MPI_INT, from, LENG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
164
165     for (int i = 0; i < len; i++) {
166         int data_len;
167         MPI_Recv(&data_len, 1, MPI_INT, from, LENG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
168         char* rdata = new char[data_len + 1];
169         rdata[data_len] = '\0';
170         MPI_Recv(rdata, data_len, MPI_CHAR, from, DATA, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
171         string d(rdata);
172         output.push_back(d);
173
174         delete []rdata;
175     }
176 }
177
178 void reduce(vector<string> &input, vector<string> &output) {
179     string previous("NULL");
180     string edges("NULL");
181     string distance("Integer.MAX_VALUE");
182     string color("WHITE");
183
184     sort(input);
185     for (int k = 0; k < input.size(); k++) {
186         vector<string> node = parseNode(input[k].c_str());
187         string node_id = node[0];
188         if (node_id.compare(previous) != 0) {
189             if (k > 0) {
190                 string nnode;
191                 createNode(nnode, previous, edges, distance, color);
192                 output.push_back(nnode);
193             }
194             previous = node_id;
195             edges = "NULL";
196             distance = "Integer.MAX_VALUE";
197             color = "WHITE";
198         }
199
200         if (node[1].compare("NULL") != 0) {
201             edges = node[1];
202         }
203         minDist(distance, distance, node[2]);
204         maxColor(color, color, node[3]);
205     }
206
207     if (input.size() > 0) {
208         string nnode;
209         createNode(nnode, previous, edges, distance, color);
210         output.push_back(nnode);
211     }
212 }
213
214 int BFS::run(int argc, char* argv[]) {
215     int size, rank;
216     MPI::Init(argc, argv);
217     size = MPI::COMM_WORLD.Get_size();
218     rank = MPI::COMM_WORLD.Get_rank();
219
220     const int FINISH = -1;
221
222     // Read data and distribute
223     vector<string> input;
224     if (rank == 0) {
225         FILE* file;
```

```
226         size_t total_size;
227         const char* name = "graph.txt";
228         LineReader* reader;
229 
230         if (argc >= 3)
231             name = argv[2];
232 
233         file = fopen(name, "r+");
234 
235         struct stat filestatus;
236         stat(name, &filestatus);
237         total_size = filestatus.st_size;
238 
239         reader = new LineReader(file, total_size);
240         char* line = NULL;
241         reader->readLine(line);
242 
243         int id = 0;
244         while (line != NULL) {
245             if (id == 0) {
246                 string node(line);
247                 input.push_back(node);
248             }
249             else { // send this data to worker process
250                 int len = strlen(line);
251                 MPI_Send(&len, 1, MPI_INT, id, LENG, MPI_COMM_WORLD);
252                 MPI_Send(line, strlen(line), MPI_CHAR, id, DATA, MPI_COMM_WORLD);
253             }
254 
255             delete []line;
256             reader->readLine(line);
257 
258             id = (id + 1) % size;
259         }
260         for (int i = 1; i < size; i++) {
261             MPI_Send((void *) &FINISH, 1, MPI_INT, i, LENG, MPI_COMM_WORLD);
262         }
263 
264         fclose(file);
265         reader->close();
266         delete reader;
267     } else { // receive data from server
268         int len;
269         while (true) {
270             MPI_Recv(&len, 1, MPI_INT, 0, LENG, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
271             if (len == FINISH) break;
272 
273             char* line = new char[len + 1];
274             line[len] = '\0';
275             MPI_Recv(line, len, MPI_CHAR, 0, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
276             string node(line);
277             input.push_back(node);
278             delete []line;
279         }
280     }
281 
282     // Process: map data, sort, exchange, merge
283     bool newLoop = true;
284     int m = 0;
285     vector<string>* other_data_tmp = new vector<string>[size];
286     vector<string>* other_data = new vector<string>[size];
287     while (newLoop) {
288         m++;
289         if (rank == 0)
290             printf("Iteration %d\n", m);
291         newLoop = false;
292         // Step 1: map data
293         for (int i = 0; i < size - 1; i++)
294             other_data[i].clear();
295         int pack = 0;
296         for (int i = 0; i < input.size(); i++) {
```

```
297             vector<string> node = parseNode(input[i].c_str());
298             if (node[3].compare("GRAY") == 0) {
299                 vector<string> edges = split(node[1].c_str(), ',');
300                 for (int i = 0; i < edges.size(); i++) {
301                     string new_node;
302                     string newDist;
303                     increaseDist(newDist, node[2]);
304                     createNode(new_node, edges[i], "NULL", newDist, "GRAY");
305                     storeNewNode(size, edges[i], new_node, other_data_tmp);
306                 }
307                 string oldNode;
308                 createNode(oldNode, node[0], node[1], node[2], "BLACK");
309                 storeNewNode(size, node[0], oldNode, other_data_tmp);
310                 pack += edges.size();
311             } else
312                 storeNewNode(size, node[0], input[i], other_data_tmp);
313             pack++;
314             if (pack >= 1500000) {
315                 pack = 0;
316                 for (int i = 0; i < size; i++) {
317                     other_data_tmp[i].insert(other_data_tmp[i].end(), other_data[i].begin(),
                            other_data[i].end());
318                     other_data[i].clear();
319                     reduce(other_data_tmp[i], other_data[i]);
320                     other_data_tmp[i].clear();
321                 }
322             }
323         }
324 
325         for (int i = 0; i < size; i++) {
326             reduce(other_data_tmp[i], other_data[i]);
327             other_data_tmp[i].clear();
328         }
329 
330         // for each phase, 1 process sends, others receive.
331         for (int phase = 0; phase < size; phase++) {
332             if (phase == rank) {// send
333                 for (int i = 0; i < size; i++) {
334                     if (i == rank)
335                         continue;
336                     sendData(i, other_data[i], other_data[i].size());
337                     other_data[i].clear();
338                 }
339             } else {// receive
340                 receive_resp(phase, other_data[rank]);
341             }
342         }
343 
344         // Step 3: Sort data
345         sort(other_data[rank]);
346 
347         // reduce phase: find the smallest distance
348         input.clear();
349 
350         string previous("NULL");
351         string edges("NULL");
352         string distance("Integer.MAX_VALUE");
353         string color("WHITE");
354         bool hasGray = false;
355         for (int k = 0; k < other_data[rank].size(); k++) {
356             vector<string> node = parseNode(other_data[rank][k].c_str());
357             string node_id = node[0];
358 
359             if (node_id.compare(previous) != 0) {
360                 if (k > 0) {
361                     string nnode;
362                     createNode(nnode, previous, edges, distance, color);
363                     input.push_back(nnode);
364 
365                     if (color.compare("GRAY") == 0)
366                         hasGray = true;
```

```
367                 }
368                 previous = node_id;
369                 edges = "NULL";
370                 distance = "Integer.MAX_VALUE";
371                 color = "WHITE";
372             }
373 
374             if (node[1].compare("NULL") != 0) {
375                 edges = node[1];
376             }
377             minDist(distance, distance, node[2]);
378             maxColor(color, color, node[3]);
379         }
380         if (other_data[rank].size() > 0) {
381             string nnode;
382             createNode(nnode, previous, edges, distance, color);
383             input.push_back(nnode);
384         }
385 
386         other_data[rank].clear();
387 
388         if (color.compare("GRAY") == 0)
389             hasGray = true;
390 
391         MPI_Allreduce(&hasGray, &newLoop, 1, MPI_BYTE, MPI_BOR, MPI_COMM_WORLD);
392     }
393 
394     string out_name("graph");
395     out_name.append(to_string(rank));
396     out_name.append(".txt");
397     ofstream out(out_name);
398 
399 
400     for (int k = 0; k < input.size(); k++) {
401         out << input[k] << "\n";
402     }
403 
404     out.close();
405     input.clear();
406     delete []other_data_tmp;
407     delete []other_data;
408     MPI::Finalize();
409     return 0;
410 }
```

B.3.3 N-Queens

Listing B.12: nQueen.cpp

```
 1 #include <mpi.h>
 2 #include "nQueen.hpp"
 3 #include <queue>
 4 #include <fstream>
 5 #include <string>
 6 
 7 #define DATA 1
 8 #define BOARD_SIZE 14
 9 
10 const int REQUEST = -1, NO_MORE_WORK = -1;
11 const int NEW = -2;
12 
13 using namespace std;
14 
15 typedef struct {
16     int size;
17     int* board;
18 } iboard;
19 
20 void push(queue<iboard>& q, int len, int* board) {
```

```
21      iboard b;
22      b.size = len;
23      b.board = board;
24      q.push(b);
25  }
26
27  bool NQueens::safe(int len, int board[], int row) {
28      int currentColumn = len;
29      for (int i = 1; i <= currentColumn; i++) {
30          int preRow = board[currentColumn - i];
31          if (preRow == row || preRow == row - i || preRow == row + i)
32              return false;
33      }
34      return true;
35  }
36
37  bool NQueens::safeAtColumn(int* board, int column) {
38      int row = board[column];
39      for (int i = 1; i <= column; i++) {
40          int preRow = board[column - i];
41          if (preRow == row || preRow == row - i || preRow == row + i)
42              return false;
43      }
44      return true;
45
46  }
47  /**
48   * Master piece
49   */
50  void NQueens::nqueen_master(int nWorker, long& total) {
51      int msg, workerid, size;
52      int* board;
53      MPI_Status status;
54      queue<iboard> workQueue;
55      queue<int> freeWorker;
56      push(workQueue, 0, NULL);
57      long unsigned int remaining;
58      bool listen = true;
59
60      //breadth first search for first 5 columns
61      while (listen) {
62          MPI_Recv(&msg, 1, MPI_INT, MPI_ANY_SOURCE, DATA, MPI_COMM_WORLD, &status);
63          workerid = status.MPI_SOURCE;
64
65          if (msg == REQUEST) {
66              remaining = workQueue.size();
67              if (remaining > 0) {
68                  iboard b = workQueue.front();
69                  MPI_Send((void*) &NEW, 1, MPI_INT, workerid, DATA, MPI_COMM_WORLD);
70                  MPI_Send(&b.size, 1, MPI_INT, workerid, DATA, MPI_COMM_WORLD);
71                  if (b.size > 0)
72                      MPI_Send(b.board, b.size, MPI_INT, workerid, DATA, MPI_COMM_WORLD);
73                  workQueue.pop();
74                  delete []b.board;
75              } else {
76                  // store free worker, then assign work for it later
77                  freeWorker.push(workerid);
78                  if (freeWorker.size() == nWorker) // all workers are free
79                  {
80                      listen = false;
81                  }
82              }
83
84          } else if (msg == NEW) {
85              MPI_Recv(&size, 1, MPI_INT, workerid, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
86              board = new int[size];
87              MPI_Recv(board, size, MPI_INT, workerid, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
88              if (size < board_size) {
89                  // send this new work to free worker
90                  if (freeWorker.size() > 0) {
91                      int id = freeWorker.front();
```

```
92                      MPI_Send((void*) &NEW, 1, MPI_INT, id, DATA, MPI_COMM_WORLD);
93                      MPI_Send(&size, 1, MPI_INT, id, DATA, MPI_COMM_WORLD);
94                      MPI_Send(board, size, MPI_INT, id, DATA, MPI_COMM_WORLD);
95                      delete []board;
96                      freeWorker.pop();
97                  } else { // store work and assign when requested
98                      iboard b;
99                      b.size = size;
100                     b.board = board;
101                     workQueue.push(b);
102                 }
103             }
104         }
105     }
106
107     for (int i = 1; i <= nWorker; i++) {
108         // stop signal to worker
109         MPI_Send((void *) &NO_MORE_WORK, 1, MPI_INT, i, DATA, MPI_COMM_WORLD);
110     }
111
112     long count = 0;
113     MPI_Reduce(&count, &total, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);
114 }
115
116 void NQueens::nqueen_worker(int rank, long& total) {
117     int msg, size;
118     long count = 0;
119     int* board;
120     string name("nqueen");
121     name.append(to_string(rank));
122     name.append(".txt");
123     ofstream out(name);
124     double t = 0.0;
125     while (true) {
126         // send work request
127         MPI_Send((void*) &REQUEST, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD);
128         // receive work
129         MPI_Recv(&msg, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
130         if (msg == NO_MORE_WORK) {
131             break;
132         }
133
134         int newSize;
135         MPI_Recv(&size, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
136         if (size < 5)
137             newSize = size + 1;
138         else newSize = board_size;
139         board = new int[newSize];
140         if (size > 0)
141             MPI_Recv(board, size, MPI_INT, 0, DATA, MPI_COMM_WORLD, MPI_STATUSES_IGNORE);
142
143         //add a new queen, breadth first search
144         if (size < 5) {
145             for (int row = 0; row < board_size; row++) {
146                 if (safe(size, board, row)) {
147                     board[size] = row;
148                     if (newSize == board_size) { // found complete solution
149                         if (store > 0) {
150                             for (int i = 0; i < newSize - 1; i++) {
151                                 out << board[i] << "-";
152                             }
153                             out << board[size] << "\n";
154                         }
155                         count++;
156                     } else {
157                         MPI_Send((void*) &NEW, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD);
158                         MPI_Send(&newSize, 1, MPI_INT, 0, DATA, MPI_COMM_WORLD);
159                         MPI_Send(board, newSize, MPI_INT, 0, DATA, MPI_COMM_WORLD);
160                     }
161                 }
162             }
```

```
163         } else {// depth first search: find the complete board
164             int column = size; // column 'size' + 1
165             board[column] = -1;
166             while (column >= size) {
167                 int row = -1;
168                 do {
169                     row = board[column] = board[column] + 1;
170                 } while (row < board_size && !safeAtColumn(board, column));
171                 if (row < board_size) {
172                     if (column < board_size - 1) {
173                         board[++column] = -1;
174                     } else { // found the board
175                         if (store > 0) {
176                             for (int i = 0; i < newSize - 1; i++) {
177                                 out << board[i] << "-";
178                             }
179                             out << board[newSize - 1] << "\n";
180                         }
181                         count++;
182                     }
183                 } else {
184                     column--;
185                 }
186             }
187 
188         }
189         delete []board;
190     }
191     out.close();
192 }
193 
194 int NQueens::run(int argc, char* argv[]) {
195     board_size = BOARD_SIZE;
196     store = 1; // store data
197     int size, rank;
198     long total;
199     MPI::Init(argc, argv);
200     size = MPI::COMM_WORLD.Get_size();
201     rank = MPI::COMM_WORLD.Get_rank();
202 
203     if (argc >= 3)
204         board_size = atol(argv[2]);
205     if (argc == 4)
206         store = atol(argv[3]);
207 
208     if (rank == 0) {
209         nqueen_master(size - 1, total);
210     } else {
211         nqueen_worker(rank, total);
212     }
213 
214     MPI::Finalize();
215     return 0;
216 }
```

B.3.4 Summation

Listing B.13: sum.cpp

```
 1 #include "sum.h"
 2 #include <iostream>
 3 #include "mpi.h"
 4 #include <cmath>
 5 #include <gmp.h>
 6 
 7 using namespace std;
 8 
 9 
10 int Sum::sum(int argc, char* argv[]) {
11     int rank, size;
12 
```

```
13      MPI::Init(argc, argv);
14      size = MPI::COMM_WORLD.Get_size();
15      rank = MPI::COMM_WORLD.Get_rank();
16
17      int startTime = MPI::Wtime();
18
19      mpz_t max;
20      if (argc == 3) {
21          mpz_init_set_str(max, argv[2], 10);
22      } else {
23          mpz_init_set_str(max, "1000", 10);
24      }
25
26      mpz_t nTasks, pSize, start, end, sum, m_rank;
27      mpz_init_set_ui(nTasks, size); // nTasks = size;
28      mpz_init(end);
29
30      mpz_init(pSize);
31      mpz_cdiv_q(pSize, max, nTasks); // pSize = max / nTasks
32
33      mpz_init(start);
34      mpz_mul_si(start, pSize, rank); // start = pSize * rank
35
36      if (rank == size - 1) {
37          mpz_set(end, max);
38      } else {
39          mpz_mul_si(end, pSize, rank + 1); // end = pSize * (rank + 1)
40          mpz_sub_ui(end, end, 1); // end = end + 1;
41      }
42
43
44      mpz_init(sum);
45      mpz_t i;
46      mpz_init_set(i, start);
47      while (mpz_cmp(i, end) <= 0) {
48          mpz_add(sum, sum, i); // sum = sum + i
49          mpz_add_ui(i, i, 1);
50      }
51
52
53      // At this point, all threads need to communicate their results to thread 0.
54      if (rank == 0) {
55          // The master thread will need to receive all computations from all other threads.
56          MPI_Status status;
57
58          for (int i = 1; i < size; i++) {
59              int sz;
60              MPI_Recv(&sz, 1, MPI_INT, i, 1, MPI_COMM_WORLD, &status);
61              char* value = new char[sz];
62              MPI_Recv(value, sz + 1, MPI_CHAR, i, 2, MPI_COMM_WORLD, &status);
63              mpz_t pSum;
64              mpz_init_set_str(pSum, value, 10);
65
66              mpz_add(sum, sum, pSum); // sum += pSum
67          }
68
69          int endTime = MPI::Wtime();
70          char* num = new char[mpz_sizeinbase(sum, 10) + 1];
71          mpz_get_str(num, 10, sum);
72          cout << num << endl;
73          cout << "Time: " << endTime - startTime << " seconds";
74      } else {
75          // The destination is thread 0.
76          char* t = new char[mpz_sizeinbase(sum, 10) + 1];
77          mpz_get_str(t, 10, sum);
78          int len = strlen(t);
79          MPI_Send(&len, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
80          MPI_Send(t, len + 1, MPI_CHAR, 0, 2, MPI_COMM_WORLD);
81      }
82
83      MPI::Finalize();
```

```
84      return 0;
85  }
```