4-30-2018

Manual and Automatic Translation From Sequential to Parallel Programming On Cloud Systems

Bing Li
Bing Li

MANUAL AND AUTOMATIC TRANSLATION FROM SEQUENTIAL TO PARALLEL PROGRAMMING ON CLOUD SYSTEMS

by

BING LI

Under the Direction of Yi Pan, PhD

ABSTRACT

Cloud computing has gradually evolved into an infrastructural tool for a variety of scientific research and computing applications. It has become a trend for many institutions and organizations to migrate their products from local servers to the cloud. One of the current challenges in cloud computing is running software efficiently on cloud platforms since many legacy codes cannot be executed in parallel in cloud contexts, which is a waste of the cloud's computing power. To solve this problem, we have researched ways to translate code from sequential to parallel cloud computing using three categories of translation methods: manual, automatic, and semi-automatic.

The performance of manual translation result is better than the other two types of translation's. However, it is costly to manually redesign and convert current sequential codes into cloud codes. Thus, the automatic translation of sequential codes to parallel cloud applications is one approach that could be taken to resolve the problem of code migration to a cloud infrastructure. During this research, two automatic code translators, Java to MapReduce (J2M) and Java to Spark (J2S), are developed to translate code automatically from sequential Java to MapReduce and Spark applications. A semi-automatic translation method is proposed, which is the combination of manual and automatic translation and performs well on large amounts of data with small fragment sizes.

This dissertation provides details about our sequential to parallel cloud code translation research in last four years. The experimental results not only indicate that translators can precisely translate a sequential Java program into parallel cloud applications but also show that it can speed up performance. We expect that an almost linear rate of speedup is possible when processing large datasets. However, some constraints still need to be overcome so more features can be implemented in future work. It is believed that our translators are the ideal models for code migration and will play an important role in the transition era of cloud computing.

INDEX WORDS: Java, Hadoop, Spark, MapReduce, Cloud Computing, Code Migration, Algorithm, Bioinformatics

MANUAL AND AUTOMATIC TRANSLATION FROM SEQUENTIAL TO PARALLEL PROGRAMMING ON CLOUD SYSTEMS

by

BING LI

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2018

MANUAL AND AUTOMATIC TRANSLATION FROM SEQUENTIAL TO PARALLEL PROGRAMMING ON CLOUD SYSTEMS

by

BING LI

DEDICATION

I would like to show my gratitude to my family and all friends for their support and understanding, especially my wife Zhewen Song for her comments that have greatly improved the manuscript.

ACKNOWLEDGMENTS

This research was supported by the department of Computer Science and the Molecular Basis of Diseases (MBD) research program at Georgia State University. I thank my research advisor Dr. Yi Pan and all dissertation committee members, Dr. Anu Bourgeois, Dr. Raj Sunderraman, Dr. Zhipeng Cai, and Dr. Mark Keil, for their advice and comments that significantly improved the thesis and all colleagues from my research group who provided insight and expertise that greatly assisted the research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

Cloud computing is a new architectural model that provides sufficient computing resources (hardware and software) to users economically. It has proven very successful in big data applications such as database applications and search engines. The fundamental step is to move jobs from conventional servers to shared cloud servers. Cloud servers provide superior services to users than those provided by personal computers or small servers [1]. The two major advantages of cloud computing are scalability and efficiency. They bring more powerful and economical computing models to users because of their large scale and resource sharing. Users only need to pay for the time and the assigned computing resources rather than purchasing expensive hardware equipment.

All users can benefit from deploying their applications on cloud server because of the cheaper costs of computing and storage resources. At the same time, the applications implementing parallel computing techniques or consuming large data storage can take full advantage of the scalability of cloud computing. Therefore, more institutions and organizations have transferred their work to cloud servers, which has led to an increasing demand for software engineers who are skilled in cloud computing [2]. Expanding the horizon of cloud computing systems so more applications can be executed in the cloud has become a major challenge.

Although parallel programming models have been studied on many different platforms, research on programming models on cloud systems is still in its infancy. Implementing parallel programs on a cloud server efficiently has always been an issue [3]. Programs developed for conventional platforms lack the parallel capability of cloud platforms. Three feasible solutions for the transition to cloud computing are mirror copying, de novo development, and automatic translation.

Mirror copying is a simple method to deploy local applications to a cloud system. Apparently, its migration cost is low, but its performance can barely be improved because it does not take advantage of the parallel architecture. Many companies and institutions have moved their systems from small private computer clusters to public cloud servers using the mirror copying method, which is the easiest way, by taking simple migration steps to reduce labor consumption. However, one significant benefit of migrating from one small private server to a larger public cloud server is more hardware resources can be allocated. The pay-as-you-go payment method of cloud services improves processing ability without the need for maintaining large, complicated equipment. However, the mirror copying method limits program scalability because these programs still work in the original mode, which is the single thread or non-scalable parallel mode.

De novo development, or the manual translation method, can effectively overcome the limitation of the mirror copying approach but is the most expensive in labor costs. It is the opposite of the mirror copying method and has high resource utilization and migration costs. Developers must refactor or even redesign original programs, and there is almost no quick way to do so, such as keeping the top program architecture and merely replacing the bottom code libraries, because of the differences between sequential and parallel modes. Therefore, besides labor consumption, the cost of training currently-employed developers is also enormous.

Automatically translating the targeted code to the desired cloud computing model is a feasible method of code migration which can simultaneously reduce migration costs and avoid resource utilization limitations. It is a middle point between mirror copying and the de novo method. Considering the huge demand for cloud computing, many institutions move their products from local equipment to cloud servers. Whether choosing to hire new engineers or retrain current employees, employers will spend a lot to transform their current products so they will be compatible with cloud computing. Apparently, if a new way is invented to automatically convert existing local programs to cloud programs, it would be well received in the cloud computing community.

This research focuses on sequential to parallel cloud computing code translation, especially in three categories: manual translation (de novo), automatic translation, and semi-automatic. The motivation of this research is to find the efficient and economical ways of code migration from sequential to parallel computing on the cloud system. Two automatic code translators, Java to MapReduce (J2M) and Java to Spark (J2S) translators, are the major contributions in this paper. There is no code translator offering sequential to parallel cloud computing code translation service. The J2M and J2S translators are the prototypes of this type of translator and have excellent performance results in evaluation experiments. The details of my research will be described in following chapters.

Our research started from a manual translation application that is a bioinformatics research project which includes a bioinformatics researcher as a collaborator. The detail of this project is demonstrated in chapter two. Inspired by this manual translation project, two automatic translators, Java to MapReduce and Java to Spark, were developed to reduce the costs of code migration. They are introduced in chapters three and four, respectively. Finally, a semi-automatic translation method is proposed in chapter five to improve the capabilities of these two translators.

2 MANUAL TRANSLATION

Cloud computing techniques can efficiently improve the efficiency of researching data- and computation-intensive problems. However, many researchers do not have the necessary programming skills to develop a cloud-based application. One possible way to implement cloud computing techniques in research projects is to collaborate with other skilled researchers. In the past several years, I have cooperated with many researchers who have no skills to help them apply cloud computing tools. This chapter demonstrates the CGI investigation project [4] as an example and describes how we can use cloud computing techniques to improve performance.

2.1 Problem

Since sequencing costs have decreased significantly, billions of human genomes could be sequenced in the next decade [5]. The enormous volume of genome sequencing data increases the difficulty of biological analysis because conventional computing resources are not suitable for handling large amounts of biological data. However, emerging cloud computing platforms such as Apache Spark provide efficient and economical ways for researchers to analyze large amounts of data. Recently, many computational biology researchers have benefited from implementing cloud computing tools in big data analytics [6] [7]. In this study, a cloud-assisted solution based on the Apache Spark framework has been applied to redefine and investigate CpG islands, which are essential epigenetic markers for biological processes in DNA genome sequences.

2.1.1 CpG Island

CpG islands are important structures in DNA genomes, and they are densely contained in specific genome regions. A CpG site refers to the area where nucleic acids cytosine (C) and guanine (G) occur together along a DNA sequence in the 5' to 3' direction, and p is a phosphate molecule located between dinucleotides. Compared with fifteen other combinations, CpG content is the lowest in DNA genomes, which is typically only around 1% [8]. Scientists have found that many biological processes appear closely related to a high density of CpG contents. In vertebrates, DNA methylation usually occurs in CpG islands, and adding methyl to cytosine can hinder transcription and silence genes. This operation can further trigger gene regulatory differentiation and epigenetic issues. In mammalian genomes, CpG islands are crucial markers when identifying biological elements and processes which play essential roles in gene regulation, gene mutation, DNA methylation, chromosome inactivation, epigenetic inheritance, and nucleosome retention [9]. The computational method for investigating CpG islands is fundamental and must be efficient in many biological studies because conventional bisulfite modification-based methods and some new whole-genome sequencing techniques are time-consuming [10] [11].

The definition of CpG islands is required to constrain the computing procedure in the computational detection of CpG islands. Generally, CpG islands are termed CpG-rich regions where epigenetic processes are closely related to the methylation status. The definition of CpG islands has three widely accepted criteria: (1) the general length of a CpG island is larger than 200 nucleotides, (2) the ratio of the observed and expected CpG content is $\geqslant 0.6$ [12], and (3) %G+C content is $\geqslant 50\%$.

These threshold-based criteria are fundamental for most existing computational CpG island prediction methods, but they have also caused CpG islands deviation in many experiments [10] [13]. Experiments verified that these criteria are violated in many cases, such as when the length of a CpG island is between eight and a few thousand nucleotides, and the percentage of G+C can be less than 50% [14]. Thus, the present threshold-based definition of a CpG island must be modified, and the detection of CpG islands is not merely a statistical task. The redefinition should comprehensively consider these existing problems since many unrevealed rules could be hidden in the CpG-enriched regions.

Since the location of genome-wide CpG islands is relevant to the chromatin structure of nucleosomes H3K4me3 and H3K27me3, a CpG island's structure could play a fundamental role in establishing chromatin structures in the pluripotent genome [15]. Thus, a CpG island's structure may contain some specific patterns that are hidden in numerous genome sequences and, therefore, are not visible. Since a CpG island is defined using coarse granularity, it is necessary to re-examine and investigate a new definition. Therefore, to take advantage of biological big data techniques, the Markov model and CpG box are introduced in our examination and investigation.

Theoretically, if the length of a genome sequence is n, the total number of possible structures is 4^n because each location in the genome sequence has four possible nucleotides. Due to the limitations of current computing techniques, thoroughly exploring such an enormous number of genome sequence structures is an impossible mission. Therefore, applying computing algorithms to reduce computing space is a feasible approach when studying the structure of a large genome sequence.

A CpG box means that a sequence which starts and ends with CpGs, as well as a CpG island, can be viewed as a set of CpG boxes. Therefore, a CpG island's structure can be measured by investigating CpG boxes. This research was inspired by analyzing the distance between CpGs [14] [16] [17]. Since a distance analysis outlines the properties of DNA's primary structures and appearance, and the CpG box also includes the features of the primary structure, it is easy to apply the Markov chain model to the re-examination and investigation of CpG islands.

2.1.2 Spark

The Apache Spark platform is an extremely fast cluster computing platform which supports multiple programming languages including Scala, Java, Python, and R. Compared with Hadoop, it has more powerful functionality and provides an efficient and robust platform for developers. Moreover, Spark is a complete processing system for large-scale parallel distributed computing integrated with multiple functionalities such as job scheduling, fault recovery, and memory management. Implementing proposed computing models on Spark can reduce the complexity of the system design and benefit from its efficiency.

The resilient distributed dataset (RDD) is the core of the Spark platform, and it is a distributed, immutable, and fault-tolerant data set. RDD is an all-purpose data container that supports any data type. It can be generated from loading an external dataset or distributing a collection from the Spark driver. Two types of operations can be handled on the RDD: transformations and actions. Transformation operations are functions processed on the RDD such as mapping, folding, and filtering. As a result, this type operation will generate a new RDD. Due to the lazy mechanism of Spark, the transformation operation is passive, so instead of being processed immediately, transformation operations only pre-configure processes. An action operation such as count, collect, and first is a signal to execute all pre-configured processes and retrieves results from the Spark server.

Unlike other computing platforms, Spark provides the in-memory mode that caches all intermediate data in the memory space. This mechanism significantly improves Spark's processing performance primarily when the intermediate data set will be reused, such as the proposed CpG project. However, as a trade-off, Spark consumes the server's memory resources. Therefore, a developer should analyze memory consumption before implementing the Spark platform.

In this project, a Scala-based Spark pipeline was developed for epigenetic analysis. The following parts of this chapter provide the details and evaluation of this project.

2.2 Method

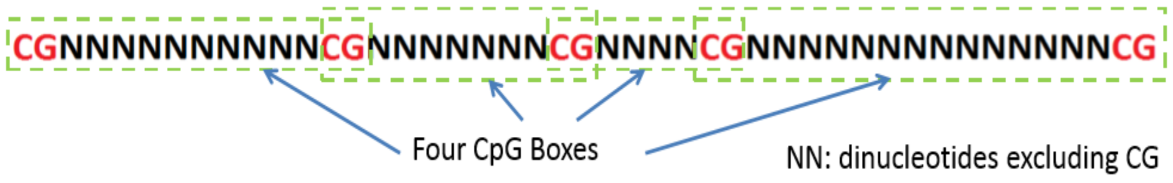
CGNNNNNNNNNNCGNNNNNNNNCGNNNNCGNNNNNNNNNNNNNNNNCG
Four CpG Boxes
NN: dinucleotides excluding CG

Figure 1 A simple example to illustrate the definition of a CpG box.

2.2.1 CpG Box

A CpG box is the area between two neighboring CpGs, and all nucleotides within the CpG dinucleotides are encapsulated in this black box. Figure 1 is an illustration of the definition of a CpG box. Because of the high mutation and maximum enthalpy [18] rate in CpG islands, a CpG box contains some biological information from the biochemical perspective. The CpG distance proposed in [14] is a property of a CpG box, which is about the length of a CpG box, while a CpG box is an object used to analyze the features of a CpG island. Therefore, we decided to study CpG boxes from the statistics of CpG distances in the human genome. In this project, large genome sequences were split into many small CpG boxes to reduce time and complexity.

The distance-based methods from [14] [16] study the sequence properties of the primary structure between any two neighboring CpG islands, which introduced a new way to understand the phenomena of CpG islands. Inspired by their results, CpG boxes are proposed in the project as a measure for CpG island investigation.

2.2.2 Assumption and Notation

CpG boxes can be analyzed as a stochastic sequence by applying the Markov property. The state of any nucleotide depends solely on the state of the previous one. It can be described in Equation 1.

$$\mathrm{P}(X_n = x_n | X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \ldots, X_1 = x_1) = \mathrm{P}(X_n = x_n | X_{n-1} = x_{n-1})$$

Equation 1

Where x_n is the nucleotide at position *n*, and X_n is the stochastic process of position *n*.

The stochastic process always starts from a CpG dinucleotide and stops at another one in a CpG box, and no additional CpGs occur in the same box. Therefore, in a CpG box with length *n,* x_1, x_2, x_{n-1}, and x_n are fixed to nucleotide C or G. Thus, if the probability of a CpG box's state is *P* and the length of this box is *n*, $P(X_1, X_2, \ldots, X_n) = P(X_2, \ldots, X_{n-1})$ because x_3 and x_{n-2} are related to x_2 and x_{n-1}. According to this property, we redefined $X_1$to the position of nucleotide

G of the starting CpG site and X_n to the position of nucleotide C of the ending CpG site for computing convenience.

Each dinucleotide can be determined by its neighbors because they have common nucleotides, and from the energy analysis viewpoint, neighboring nucleotides constitute the structure of a CpG box [19]. Therefore, dinucleotides can be studied in the chain model.

2.2.3 Markov Chain Model

Equation 2 represents the Markov chain model.

$$P(X_n = x_n, X_{n-1} = x_{n-1}, \dots, X_1 = x_1) =$$
$$P(X_n = x_n | X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_1 = x_1)$$
$$\times\ P(X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_1 = x_1)$$

Equation 2

Equation 3 is the result of applying Equation 1 to Equation 2.

$$\begin{aligned} P(X_n = x_n, X_{n-1} = x_{n-1}, \dots, X_1 = x_1) \\ &= P(X_n = x_n | X_{n-1} = x_{n-1}) \times P(X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_1 = x_1) \\ &= P(X_1 = x_1) \prod_{i=2}^{n} P(X_i = x_i | X_{i-1} = x_{i-1}) \end{aligned}$$

Equation 3

Since the nucleotide is fixed, $P(X_1 = x_1) = 1$. According to the Bayesian equation, Equation 3 can be further induced to Equation 4.

$$P(X_n = x_n, X_{n-1} = x_{n-1}, \dots, X_1 = x_1) = \prod_{i=2}^{n} \frac{P(X_i = x_i, X_{i-1} = x_{i-1})}{P(X_{i-1} = x_{i-1})}$$

Equation 4

Since X_i and X_{i-1} , $i \in \{2, \dots, n\}$ are neighbors, the dinucleotide $D_{i-1}$is introduced to replace X_i and X_{i-1}. Equation 5 is the result of this simplification.

$$P(X_n = x_n, X_{n-1} = x_{n-1}, X_{n-2} = x_{n-2}, \dots, X_1 = x_1) = \prod_{i=2}^{n} \frac{P(D_{i-1})}{P(X_{i-1} = x_{i-1})}$$

Equation 5

CpG boxes fit the Markov chain model well for two reasons. First, it constrains the stochastic walk. The length of each CpG box is in a specific range that is not too large. Second, the length of a CpG box varies depending on its biochemical structures. This means a CpG box contains some biochemical meaning and is not random. Therefore, the length of each CpG box is related to the length of its neighbors. The probability of length variation in neighboring CpG boxes is discussed in the following section.

2.2.4 Maximum Likelihood

Regarding Equation 5, for all CpG boxes inside CpG islands, the probabilities of each dinucleotide located at a specific position can be acquired, and the probabilities of dinucleotides in each type of CpG box can also be obtained for the CpG boxes outside of the CpG islands.

θ is the estimation parameter, θ ∈ Θ, Θ is the set of $\{\theta_{cgi}, \theta_{noncgi}\}$, simplified as $\{\theta_C, \theta_N\}$. $L(\theta;\ X_n, X_{n-1}, \dots, X_1)$ denotes the likelihood of an estimator with observations $X_n, X_{n-1}, \dots, X_1$ and a parameter θ. Equation 6 is the result of applying the Bayesian theorem.

$$P(\theta|X_n, X_{n-1}, \dots, X_1) = \frac{P(X_n, X_{n-1}, \dots, X_1|\theta)P(\theta)}{P(X_n, X_{n-1}, \dots, X_1)}$$

Equation 6

When the parameter θ is under the uniform prior distribution, the maximum likelihood estimator is the same as the most probable Bayesian estimator. Specifically, $P(\theta)$ in Equation 6 is under uniform distribution, and $P(X_n, X_{n-1}, \dots, X_1)$ and θ are independent. Therefore, Equation 6 can be induced to Equation7.

$$L(\theta; X_n, X_{n-1}, \dots, X_1) \equiv P(\theta|X_n, X_{n-1}, \dots, X_1) \propto P(X_n, X_{n-1}, \dots, X_1|\theta)P(\theta)$$
$$\propto P(X_n, X_{n-1}, \dots, X_1|\theta)$$

Equation 7

2.2.5 Neighboring CpG Boxes

Examining neighboring CpG boxes is a necessary procedure when investigating CpG islands because they consist of sequential CpG boxes. Equation 8 states the likelihood of estimator θ for $B_i \rightarrow B_{i+1}$. Like Equation 7, the maximum likelihood estimator is equal to the most probable Bayesian estimator when the parameter θ is under uniform prior distribution. Since there is no estimator probability detected for non-CGI or CGI, the probability of estimator θ is assumed to be under uniform distribution, and $P(B_i \rightarrow B_{i+1})$ and θ are independent.

$$\begin{aligned} L(\theta; B_i \rightarrow B_{i+1}) &\equiv P(\theta | B_i \rightarrow B_{i+1}) \\ &= \frac{P(B_i \rightarrow B_{i+1} | \theta)}{P(B_i \rightarrow B_{i+1})} \\ &\propto P(B_i \rightarrow B_{i+1} | \theta) P(\theta) \\ &\propto P(B_i \rightarrow B_{i+1} | \theta) \end{aligned}$$

Equation 8

2.3 Design and Implementation

From the computer science perspective, this project is a traditional data-intensive statistical problem. Since the size of all sample genome sequences is larger than two gigabytes in this experiment, the total data size is enormous. Thus, the hardware needs an extensive memory capacity and a considerable amount of time will be required. To reduce time and equipment costs, parallel and cloud computing techniques are implemented in this project.

This problem can be easily converted to a data independent model because the dependency only appears between neighboring CpG boxes. For example, when calculating $P(B_i \rightarrow B_{i+1})$, both B_i and B_{i+1} are used at the same time. However, this problem can be converted to an independent model by duplicating common CpG boxes, which is shown in Figure 2.

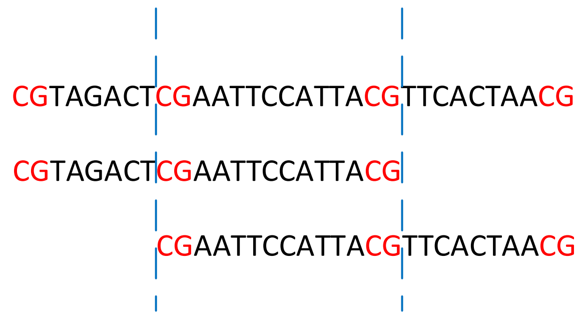

Figure 2 Dependence removed by duplicating CpG boxes

By using this mechanism, the task can be easily processed in parallel in the following steps.

2.3.1 Sequence Division

First, the original sequence needs to be trimmed and split by the CG makers. The input data format determines the process model selection, either sequential or parallel. In this experiment, the sample data files contain raw genome information, which is the entire genome sequence without any spaces, line breaks, or other additional information. In this case, files can be processed in parallel, but in each file, the data must be handled by a single process thread.

In Figure 3, a new CpG box list is generated by splitting the input sequence. Since this research only studied the length of each CpG box, the letter sequences can be replaced by their length number. This method can naturally reduce data size and improve process performance in future procedures.

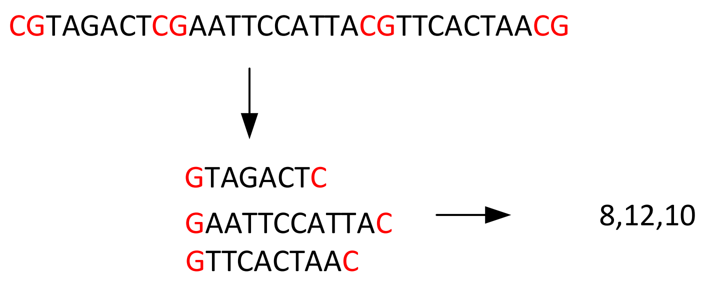

Figure 3 Sequence Division

```
//load DNA file
val source =
  spark.sparkContext.textFile(file_path)
    .flatMap(_.split("\n"))

//CG boxes
val cgBoxes = source.flatMap(_.split("CG"))
```

The code above is the Spark implementation, which is developed in the Scala programming language. There are two tasks in the sequence division step. First, since each file may contain multiple DNA sequences, the program should split these sequences using a delimiter. In this example code, the delimiter is a line break. Second, each DNA sequence should be further split by CGs to retrieve the CG boxes.

2.3.2 Data Preparation

The procedure following sequence division is converting length number sequences to neighboring CpG box couples. Figure 4 shows three steps to accomplish this task. The sequence leng is duplicated, shifted, and combined in this procedure. This procedure completes all necessary preparation and generates a length pair list for the probability calculations.

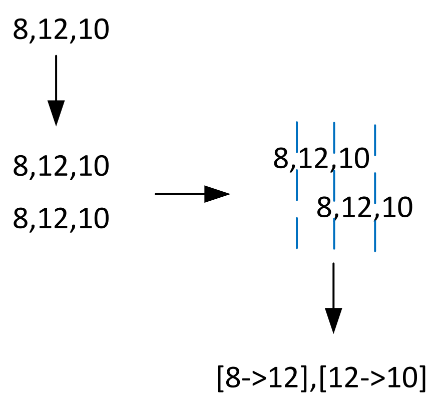

Figure 4 Data Preparation

```
//count sizes
val sizes = cgBoxes.map(_.length)

//CG jumps
val list1 = sizes.zipWithIndex().map{
  case (v,i) => (i, v)
}
val list2 = sizes.zipWithIndex().map{
  case (v, i) => (i-1, v)
}
val jumps =
  list1.leftOuterJoin(list2).flatMap{
  case (i, (a, b)) => b.map(a -> _)
}
```

This piece of code is responsible for counting the size of each CG box and grouping all neighboring sizes in DNA sequences. Like the design described previously, it will duplicate the entire sequence and group neighboring numbers.

2.3.3 Probability Calculation

The last procedure is the most straightforward step because functions from the standard Spark application interface (API) can handle these simple statistical tasks. In probability calculations, the objective is to obtain the distribution of CpG boxes of different lengths. Therefore, all length pairs need to be grouped by their length. For instance, to calculate P(3 → 5), we need to count the total sample size and how many (3 → 5) is in the length pair list. After scoring all necessary factors, the target probabilities can be calculated using simple mathematical calculations.

```
//total size
val totalSize = jumps.count()

//probability of each type of jumps
val result = jumps.groupBy(_).map{
  case(k, v:Seq[(Int, Int)]) =>
    (k, v.size.toDouble/ totalSize)
}
```

The piece of code described above is the last step of the Spark implementation. It counts the number of time each unique number pair repeats in the DNA sequence and uses these numbers to calculate the probabilities of each length variation.

2.4 Evaluation

In this project, the proposed parallel process method was used on mouse and human DNA sequences, which has 21 and 24 chromosomes, respectively. This evaluation examined three aspects of the proposed method: unit computing costs, time costs of large-scale data, and speedup on multiple cores.

2.4.1 System Configuration

The proposed method was examined on a small computer cluster that has five computer nodes. Each computer node has a dual Intel Xeon E5-2650 CPU and 64GB of DDR3(1866MHZ) memory. The Apache Spark platform's configuration is listed below.

- Total executor memory: 50GB
- Total number of executor cores: 50

2.4.2 Chromosome Test

The first experiment tested the performance of the proposed method on two different datasets, human and mouse DNA sequences. Figure 5 and Figure 6 describe the result of these two test cases. When the length of chromosomes increases, the processing time also increases. However, Figure 5c and 6c show that the time/length ratio decreased when processing the larger dataset. Specifically, the proposed method can handle large-scale datasets more efficiently than smaller ones. Note that the results shown in these two figures are not entirely linear because some datasets such as extremely large CpG boxes may increase time consumption.

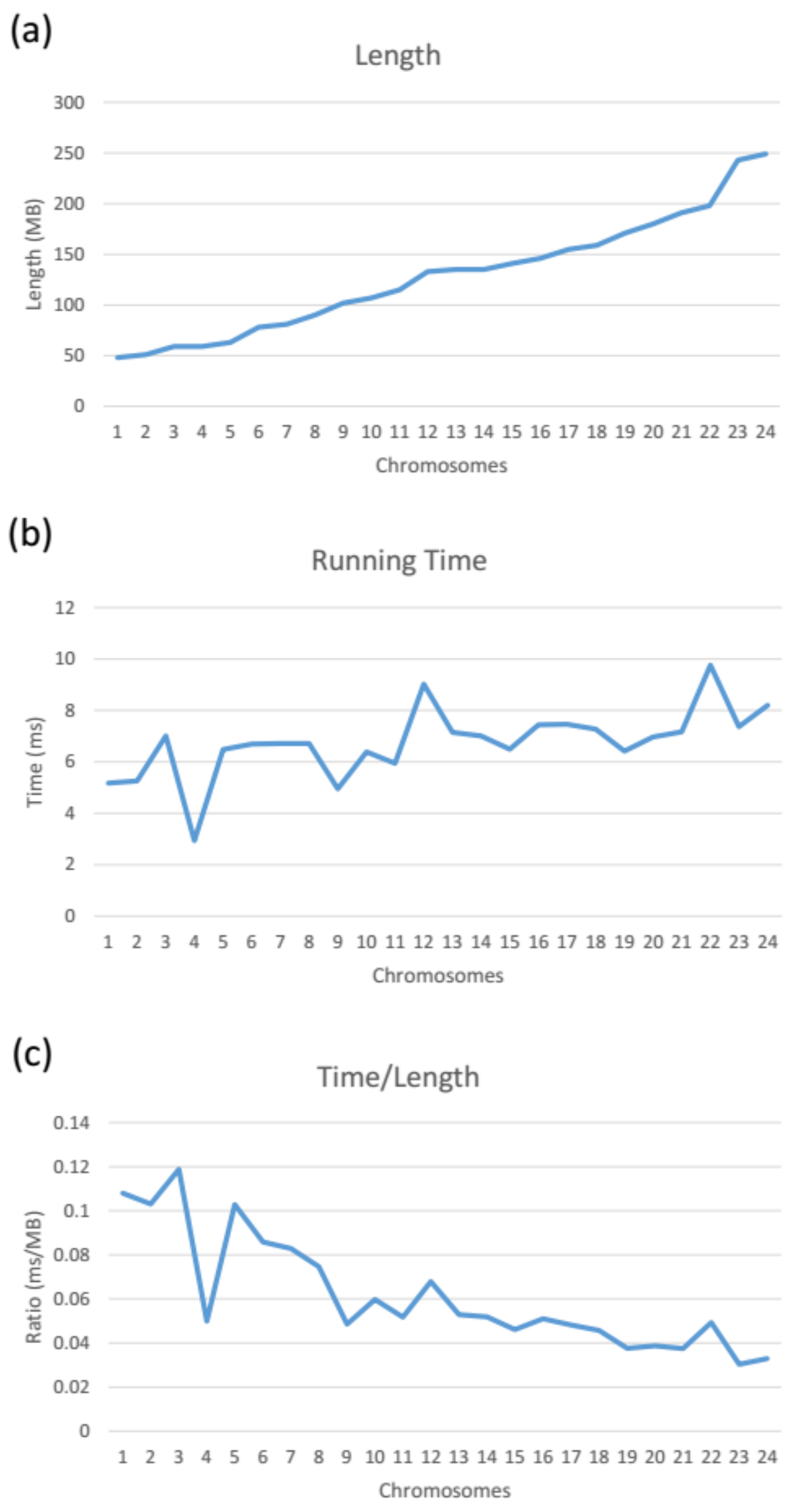

Figure 5 Running time and length of human chromosomes

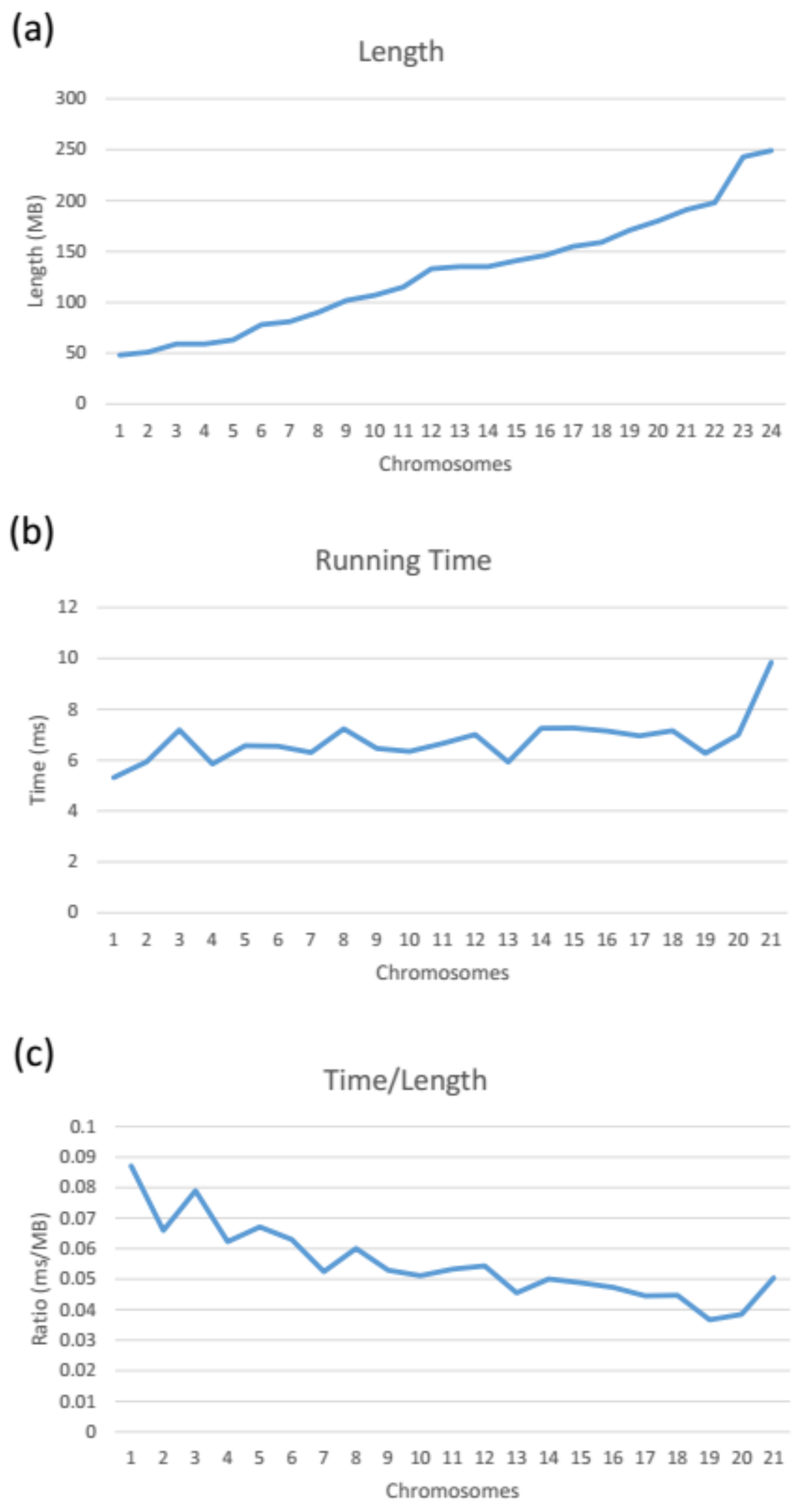

Figure 6 Running time and length of mouse chromosomes

2.4.3 Whole-Genome Test and Speedup Analysis

The second experiment evaluated the performance of the proposed method on a large-scale dataset because the first test proved that it could handle larger datasets more efficiently. The sizes of whole human and mouse genome sequences are 3 gigabytes and 2.7 gigabytes, respectively. The time consumption of these test cases is described in Figure 7.

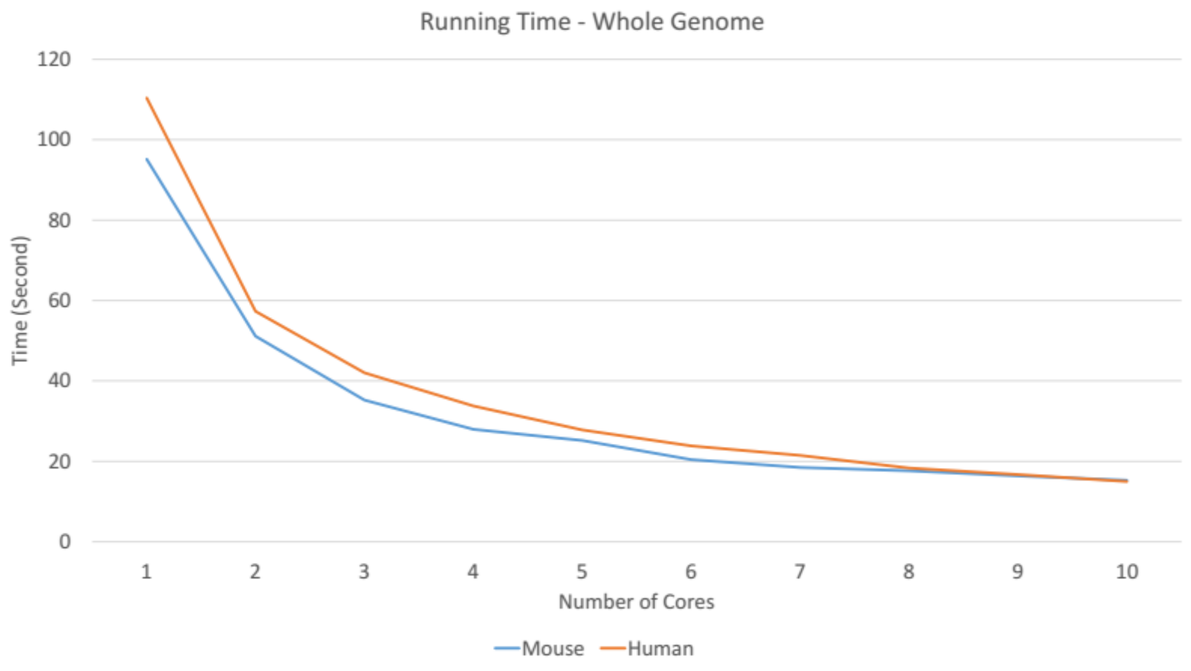

Figure 7 The correlation between running time and the number of cores using whole genomes of human and mouse DNA as an illustration.

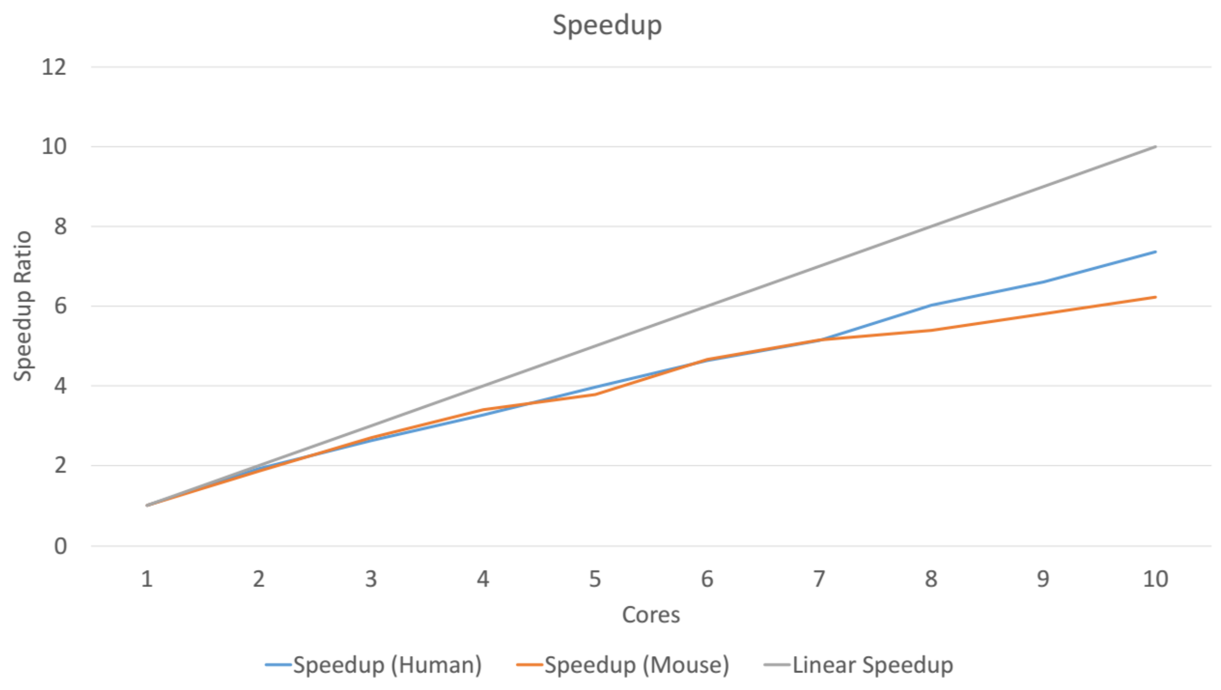

Figure 8 Speedup ratios of human and mouse genomes

Figure 8 shows the speedup rate based on the result of the whole genome test. The speedup rate increases when implemented on more processing cores, and it can be 6–7 times faster when using 10 cores. However, as the number of cores increase, there is a decrease in the speedup rate. It can be expected that the speedup rate will decrease when implementing on too many processing cores due to the overhead cost of the parallel computing technique. Therefore, users should analyze the input dataset and configure the number of cores to obtain optimal performance.

2.5 Conclusion

The CpG island project is a typical application of parallel computing and cloud computing techniques. In this project, developers can easily apply parallel computing techniques and deploy their programs on a cloud server because of the consistent data structure. Each CpG box pair can be processed individually, and the result can be obtained by combining all individual results from previous procedures. For developers experienced in parallel programming, it only takes a few minutes to design this system. However, the problem is that some researchers do not have the necessary skills.

Unlike some existing analysis software, users only need to input some configurations, and the software can handle all remaining tasks, while deploying projects on cloud computing platforms such as Apache Spark requires programming skills. It is a significant challenge to researchers without a programming background. Therefore, an interesting problem was identified during this project, which is how to reduce the complexity of developing parallel cloud computing programs.

3 JAVA TO MAPREDUCE TRANSLATION

Researchers want to benefit from cloud computing techniques without spending too much time and money on programming. Requesting help from experienced developers is not a good solution because it is also expensive. Besides the monetary expenses, they need to spend a lot of time helping developers understand their project's background. A misunderstanding may result in problems during the program's development. Therefore, it is difficult to cooperate with developers without a corresponding background. To solve this problem, it might be more efficient if researchers write programs themselves. However, it seems that the problem goes back to the origin, that most researchers do not know how to write programs.

How can researchers learn to program on cloud systems by themselves? The proposed solution is using automatic program translation tools. Java to MapReduce translator (J2M) [20] is a code migration tool that can automatically translate Java for-loops to the MapReduce program. When using this translator, users don't need to have any parallel or cloud computing experience. They just write the program in the regular for-loop format, and the translator will translate these loops to MapReduce functions that can be executed on the Hadoop platform. This translator can significantly reduce the programming difficulty of cloud applications. Thus, the researchers can program by themselves without any additional help from others.

3.1 Background

Among the many emerging programming models for cloud platforms, MapReduce has become one of the most popular for big data analysis in cloud systems due to its simplicity in implementing data parallel applications [21] such as Apache Hadoop [22], Phoenix [23], Mars [24], Twister [25], and Granules [26].

- Apache Hadoop [22] was developed for data-intensive distributed applications.
- Phoenix [23], a shared-memory implementation of MapReduce for data-intensive processing tasks, can be used to program shared-memory multiprocessors as well as multi-core chips.
- Mars [24], a GPU implementation of MapReduce, provides a generic framework for developers to implement data and computation-intensive tasks correctly, efficiently, and easily on graphics processors (GPUs). It hides the programming complexity of the GPU behind the simple and familiar MapReduce interface. Hence, developers can program on GPUs without any knowledge of the GPU's architecture or the graphics' APIs.
- Twister [25], an implementation of the iterative MapReduce model, is lightweight at runtime. It provides a feature for cacheable MapReduce tasks and allows programmers to develop iterative applications without spending much time reading and writing large amounts of data in each iteration. Twister4Azure [27], a version for Windows Azure, has also been released.
- Granules [26] is a lightweight, streaming runtime for cloud computing with support for MapReduce.

As the need to analyze vast amounts of data stored in diverse locations increases, Hadoop grows more popular. Hadoop provides a great platform for logistics and coordination such as task failures, loading inputs, splitting inputs, and task coordination. It makes the programming job easy. Each job we encounter in MapReduce presents an applicable way to resolve the problem with the amount of data increasing significantly. However, it is not easy to learn how to program in MapReduce because its paradigm is low-level and rigid. To develop applications with these runtime systems, developers should be proficient in the following three domains: the MapReduce model, runtime systems, and the system-specific programming language. Although many applications can be parallelized using this method, developers still need to be trained intensively in the three domains. Thus, the transition to cloud programming for developers can be torturous. It will be desirable if there is an alternative way to utilize the Hadoop platform without learning MapReduce programming.

3.2 Related Work

Manually converting sequential code to cloud code is a time-consuming and error-prone way to build a cloud application because the data flowing in and out of the MapReduce code must be carefully and precisely coordinated with other sequential sections of the code [28]. For example, a Hadoop programmer must write very verbose codes based on MapReduce. Through the code translator, the sequential codes can be translated to cloud codes automatically. Many X-to-MapReduce (X is a programming language) translators and relevant systems have emerged. Several SQL-like declarative languages and their code translators have been built and integrated with MapReduce to support these languages. It includes Pig Latin/Pig [29] [30], SCOPE [31], Hive/HiveQL [32], YSmart [33], YSmart-S, Jqal [34], Tenzing [35], HadoopDB [36] [37], SQL-to MangoDB, and others. In the following section, we will review some representative works.

3.2.1 Hive/HiveQL

Hive [32] was initially created by Facebook in 2007 as their data warehouse software to manage large-scale datasets in distributed systems. Based on Apache Hadoop, Hive provides a mechanism to extract, transform, and load data in different data storage systems such as Apache

HDFS and Apache HBase. The semantics of Hive is a SQL-like language, which has been called HiveQL for a relatively short time and performs MapReduce analysis not only with its built-in library but also with its extendable capability. By extending and overwriting the built-in functions, Hive can be used in more sophisticated analyses. On Facebook's servers, more than 700TB of data and tens of thousands of tables are stored in the Hive warehouse, which is utilized by over 200 users each month.

3.2.2 HadoopDB

HadoopDB [36] is a hybrid system of MapReduce and DBMS technologies. There are four main design features of HadoopDB; it is designed to run on a shared-nothing cluster of commodity machines or in the cloud, as an attempt to fill the gap in the market for a free and open source parallel DBMS, for greater scalability than currently available parallel database systems and DBMS/MapReduce hybrid systems, and to be as scalable as Hadoop while achieving superior performance in structured data analysis.

HadoopDB extends the Hadoop framework, which is the core of HadoopDB, by providing the following four components:

- Database connector

  This is the interface between TaskTracker and independent database systems residing on nodes in the cluster. It allows Hadoop jobs to access multiple database systems by executing SQL queries.
- Catalog

This maintains meta-information about the database. It contains both meta-data about the location of database chunks stored in the cluster and statistics about the data.

- Data loader

  This hash-partition splits data into smaller chunks and coordinates their parallel load into the database system.

- Query interface

  This allows queries to be submitted via a SQL/MapReduce API.

The SQL to MapReduce to SQL (SMS) planner has extended two areas of Hive: (I) updating the MetaStore with references to our database tables before any query execution and (II) performing two passes over the physical plans between the physical query plan generation and the MapReduce job execution. The prototype was built using PostgreSQL as the underlying DBMS layer in HadoopDB. It may leverage any JDBC-compliant database system due to its design. Based on HadoopDB, a new solution is given. It can transform a single-node DBMS into a highly-scalable parallel data analytics platform that can handle very large datasets and provide automatic fault tolerance and load balancing [37].

3.2.3 YSmart: SQL-to-MapReduce Translator

YSmart [33] is a SQL-to-MapReduce translator which provides a way to minimize the number of MapReduce jobs, particularly for multiple correlated operations in complex queries. Compared with other existing translators, YSmart reduces redundant computations, I/O operations, and network overhead. Some results have demonstrated that YSmart outperforms Hive and Pig on correlated queries execution.

The claimed contribution of YSmart is building a correlation-aware SQL-to-MapReduce translator to improve complex query performance without modifying the underlying Hadoop MapReduce system. MapReduce has limited functionality for complex queries. For instance, local disk copies of intermediate results are needed for node failure, and temporary results must be uploaded to the global file system. Both processes will create redundant overhead. Moreover, there is no mechanism to reuse the intermediate data between concurrent jobs. YSmart is designed to remove these limitations by detecting the correlation of complexities within queries, translating the queries into efficient MapReduce programs, and reducing the overhead in MapReduce processes. Specifically, YSmart defines four types of operations for MapReduce jobs: *selection-projection (SP), aggregation(AGG), join*, and *sort*. With these defined jobs, YSmart translates one operator to one job from the original complex queries to MapReduce programs. Significant optimization is implemented via YSmart job merging. YSmart recognizes the intra-query correlation and provides a set of rules for multiple job merging. These rules are defined to handle different cases. The details can be found in [33]. Since YSmart presents a generic mechanism to translate and optimize SQL-like language to MapReduce jobs, Hive has recently adopted YSmart as a patch.

All above three projects are SQL-to-MapReduce applications and limited to SQL query translation. The following two projects are the implementation of computational statement translation.

3.2.4 M2M Translator

The previously discussed languages and translators inspired Zhang et. al. to develop the M2M translator [38] in 2013; it can translate Matlab codes to MapReduce codes. Matlab is a popular programming language in scientific research and was developed by MathWorks. As a calculation tool, Matlab has a complete and powerful function library. Therefore, it can be easily learned and requires less programming knowledge. Moreover, users do not need to spend much time on programming. For example, users can call functions from the standard Matlab library directly when they need to operate matrices like cross products and dot products. Specifically, the main part of the Matlab program is a combination of standard functions. Therefore, the Matlab program's major calculation jobs are processed in standard library functions.

The basic idea of the M2M translator is to build a new function library implemented on cloud servers instead of the standard application programming interface. When users translate a Matlab source file, the translator can use the new functions to replace the standard API functions. This pre-translations model is effective but restricted, which means users cannot translate their own functions.

3.2.5 OpenMP

OpenMP is a collection of famous industry-level APIs for C/C++ and Fortran programming [39]. It has already become a standard for most C/C++ and Fortran compilers. When users implement OpenMP in the source code, the compiler can convert targeted portions of the code highlighted by users to multi-threads models. OpenMP was developed in 1997 before cloud computing was invented, so it cannot be applied to cloud computing program translation. Therefore, a new sequential program for cloud program translation (J2M) is proposed in this paper. It implements partial features of OpenMP and is effective at cloud program translation.

3.3 J2M Translator

The objective of the proposed translator is to translate one loop marked by users in a Java source file to a MapReduce function. This translator is similar to a typical compiler, but the J2M translator only translates partial code and the target loop while the remaining code are unchanged. Since both the source codes and the translation results are Java code, the translator does not parse the entire source file but only extracts necessary information from the source file and then generates the result.

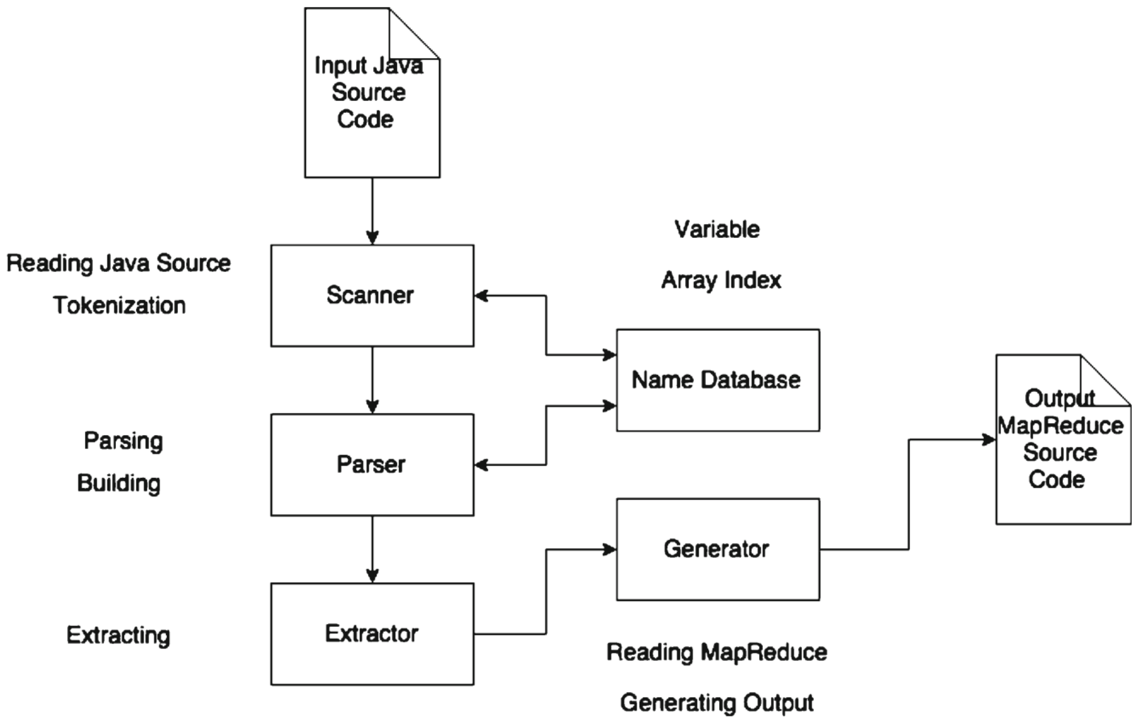

Figure 9 J2M translator structure

The structure of the proposed translator is shown Figure 9. The translator has four major components: a scanner, a parser, an extractor, and a generator. Each is responsible for a translation procedure, namely tokenization, parsing, extraction, and generation. It also has a minor component termed the name database, which can help the parser and the extractor record the variable names.

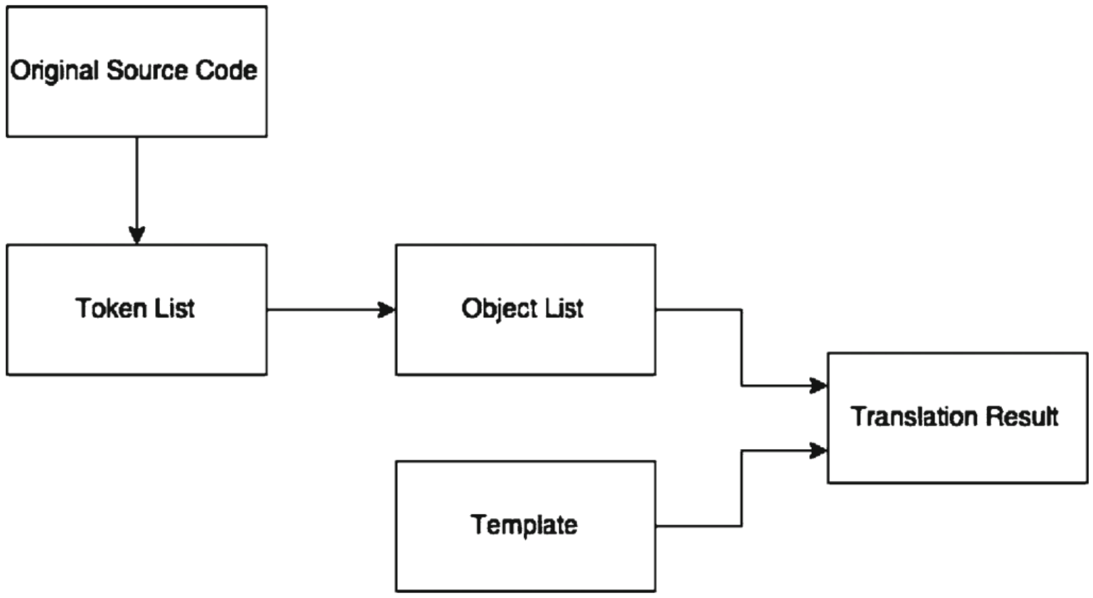

Figure 10 Translation procedures

When the translator translates a Java source file, the scanner tokenizes the entire source file, generates a token list, and then sends this list to the parser. The parser analyzes this token list and generates a new objects list, packs them, and passes this package to the generator. In the final step, the generator combines one MapReduce template with the information package received from the extractor and then generates the translation result. These procedures are shown in Figure 10.

Unlike a typical compiler, which has only two stages (i.e. tokenization and parsing), the J2M translator translates Java source files in four stages: tokenization, parsing, extraction, and generation. Next section provides details of each stage in the translation process, followed by a translation example used to illustrate each stage.

3.3.1 Tokenization

In the tokenization stage, the scanner scans the entire input Java source file and generates a token list by splitting the code into tokens. This token list is not a typical token list in a compiler where all tokens are represented by strings. Instead, in the proposed token list, it not only records all token values by string objects but also records the types of tokens, which is an additional job of tokenization.

#	**Token Types**
1	Identifier
2	Value
3	Operator
4	Directive
5	EOF

Table 1 Token List

The proposed translator has five types of tokens as represented in Table 1. In this stage, all names that represent variables, classes, methods, packages, and reserved words are categorized as an *identifier*. It is the largest family of tokens in the tokenization stage and is subsequently split into a few subgroups in the next stage for parsing.

The *value* token is a set of string values and seven Java primitive values: float, double, int, byte, long, char, and short. Another Java primitive type is Boolean. In this stage, the Boolean value will be categorized as an *identifier* and converted to a value in the parsing stage. It is not necessary to separate these values into different groups because all input Java source files should be checked by a Java compiler without compiling errors. Therefore, the translator skips the syntax check for this input source file. Another reason is the proposed translator is not responsible for analyzing values passed to the result file.

Operator tokens represent all symbols that are not only Java operators but also other syntax symbols such as a semicolon, comma, and period. Like *value* tokens, the proposed translator does not parse these symbols, which may help the translator analyze identifiers in the parsing stage.

The *directive* is a mark to highlight the target loop for the J2M translator. There are only two types of directives, *//J2MSTART* and *//J2MEND,* for start and end, respectively. Since directives use the comment format, it is not effective when this source file is compiled by a Java compiler. In the tokenization stage, all Java comments that start with *//*, */**, and **/* are to be eliminated except directives.

Only one end of file (*EOF)* token appears at the end of the token list. It is not extracted from the input Java source file but automatically generated by the scanner. It indicates that the token list is terminated.

For a real-world example, the following sequential Java code will be translated:

```
//Original Loop

//J2MSTART
for(int i = 0;i < 10;i++){

    a[i] += num;

    a[i+1] += i;
}
//J2MEND
```

At the end of tokenization stage, the token list should be as follows:

```
Token List:

…

//J2MSTART : Directive

for        : Identifier

(          : Operator

int        : Identifier

i          : Identifier

=          : Operator

0          : Value

…

EOF        : EOF
```

3.3.2 Parsing

The parsing stage is the core of translation procedures, which converts simple string tokens into more meaningful objects. For instance, in the parsing stage, the variable token is converted into the variable object that not only represents the name of a variable but also records the type information. Unlike the parsing stage in a regular compiler, analyzing the entire code or entire instructions is unnecessary. The proposed translator analyzes only the single token.

#	Objects
1	ClassName
2	Directive
3	EOF
4	Imports
5	MethodName
6	Operator
7	PackageName
8	Types
9	Value
10	Variable
11	ArrayVariable

Table 2 Object List

Compared to the scanner, the parser converts all tokens into 11 objects represented in Table 2. *Value, EOF*, and *directive* objects are the same as those in the tokenization stage. The parser simply converts them from string tokens to objects without any additional features.

Operator objects combine operators and reserved words in Java together because these two kinds of tokens have similar properties. In the parsing stage, the parser converts all *operator* tokens and reserved word tokens into Operator objects. The J2M translator records all Java reserved words in a list. When the parser analyzes *identifier* tokens retrieved from the tokenization stage, it collects all reserved words and replaces them with *operator* objects.

Type objects are extracted from *identifier* tokens, which contain two parts. One includes the eight Java primitive types: bytes, int, short, long, Boolean, float, double, and char. The other part is defined as the object type. The *parser* analyzes all object types in declaration instructions, the variable declaration and method declaration, and then converts them into *type* objects.

The *variable* object is the successor of the *identifier* token. It is declared in the declaration instruction and then used anywhere afterward. The parser needs to analyze the types of all variables, which contains two cases; one is inside the variable declaration and the other is outside of it. It is easy to extract the type of variables inside the declaration instructions because the *type* object is beside the variable object. However, it is different when it is outside of the declaration. To realize this feature, the J2M translator creates a variable stack in the parsing stage. Each time it meets a variable outside of a declaration, it searches in the variable stack to retrieve the types of that variable.

One special variable case is an array variable. In the J2M translator, the *ArrayVariable* object represents the array variable. There are two phases of an array variable: one is the array reference, identified in Java as *array_name*; the other one is the element reference, identified as *array_name[index]*. Normally, the user does not create a new array in a loop. Thus, the J2M translator does not currently support creating new arrays in the target loop. The translator transforms the loop to MapReduce function, and then the loop is split into many small pieces. To reduce the cost of data transmission, the translator only transports partial elements of the array used in the mapper function. Therefore, the translator converts these array elements to a new individual variable. For instance, *array_name[1]* can be translated as *array_name_1*.

ClassName and *MethodName* objects work together because they are both special variables. In our translation result, the method name always contains its class name. To achieve this feature, each time the parser detects a method declaration, it searches for the nearest class name in the variable stack. Once it finds the target, it sets a reference to this *ClassName* object corresponding to the *MethodName* object.

PackageName and import objects are also very important parts of codes. They contain the package name of the target class and all dependent classes. Since the translation result works in the original package, it copies these two parts to the result directly when the J2M translator generates the translation result. At the end of this stage, an example of a token list provided in the last stage is shown in the following object list:

```
Object List:
…
//J2MSTART : Directive
for       : Operator
(         : Operator
int       : Types
```

```
i       : Variable, type = int

=       : Operator

0       : Value

…

EOF     : EOF
```

3.3.3 *Extraction*

To generate the translation result, the J2M translator implements a MapReduce function template. More details about this process are described in the next section, the generation stage. In this template, four blank areas needed to be filled:

- Package name
- Import classes
- Variable declaration
- Function body

The required information is extracted from the input source code. The objective of the extractor is to prepare all necessary information for generating the final translation results.

The package name and import classes are copied from *PackageName* and imports objects directly. The variable declaration and function body are both extracted from the target loop in the source code file. There are three types of variable operations in the template. The first type is the declaration. The extractor scans the target loop and collects all variable information. Next, it simultaneously creates the declaration list and converts array variables into regular variables. It is unnecessary to declare array variables in the translation result.

The last two stages of the process are data import and export. The translation result should be executed on a Hadoop cluster, which requires that all variables are loaded from a Hadoop File System (HDFS). Therefore, in the result, variables are loaded from files after declaration and then exported at the end of this program.

Apart from variable operations, the target loop can be transformed into MapReduce functions. The transformation contains two steps: convert array variables into regular variables declared during variable declaration and move everything from the target loop to the result function.

Following the example object list in the last stage, the information can be extracted as shown below.

```
Package Name :
…
Import Classes :
…
Variable Declaration :
int i ;
int array_a_0 ;
int num ;
int array_a_1 ;
Function body :
array_a_0 += num ;
array_a_1 += i;
```

Note that the array elements in the original Java code, *a[i]* and *a[i+1],* were transformed into regular variables, *array_a_0* and *array_a_1*, in the parsing stage.

3.3.4 Generation

The generation stage is simpler than the previous three stages. The *Generator* combines the results of the extraction stage and the template to generate a final translation result code. The most important aspect is the design of MapReduce function template. The following is our template design. Due to the length limitation, the design of the template is a simplified version of the real and complicated one.

```
/*Package Name*/

/*Import Classes*/

public static class Map{

    public void map(){

        /*Variables Declaration*/
        while(/*Some Conditions*/){

            /* Body, including data import and export */

        }
    }
}
```

Compared to OpenMP, which can convert some dependent loops to independent loops, the J2M translator currently only supports independent loops. Although the first generation of the

J2M translator is still in its trial period, more practical features will be added to make it more sophisticated in the future. The J2M translator's constraints will be introduced in the results and performance section.

The J2M translator only implements the mapper function since the reducer function is not necessary for the loop translation. The mapper function executes each step of the original loop using a new while-loop. Each mapper function can execute multiple steps of the original loop. The total number of mapper functions depends on the environment variable. If the total steps of the original loop are *M* and the total number of mappers is *N*, the steps in each mapper function should be $\frac{M}{N}$. In this template, there are four p areas: package name, import classes, variable declaration, and body. In the generation stage, these four areas can be filled by information retrieved from the extractor.   The result of the example code is as follows:

```
package ...

import ...

... class and method headers ...

int i;
int array_a_0;
int num;
int array_1_1;

while(tokenizer.hasMoreTokens()){

  …
  i =
    Integer.valueOf(tokenizer.nextToken( ));

  array_a_0 =
    Integer.valueOf(tokenizer.nextToken( ));

  num =
    Integer.valueOf(tokenizer.nextToken( ));

  array_a_1 =
    Integer.valueOf(tokenizer.nextToken( ));
  …

  array_a_0 += num ;
  array_a_1 += i ;
  output.collect(
       … ,
        new Text(“ ” + i + “ ”
         + array_a_0 + “ ” + num
          + “ ” + array_a_0 ));
  …


}
```

3.4 Evaluation and Assessment

After a Java file is translated using the J2M translator, two facts are of concern to the user: computing correctness and performance speedup. One of the most important features of the translation result is performance, especially the execution speed. If the translation result's execution speed is slower than the original Java program, the proposed translator is useless. To test the performance of the J2M translator, experiments based on these two concerns were created. The experimental results not only indicate that the translator can precisely translate sequential Java into cloud codes but also show that it can increase performance speed, and we expect that an almost linear speedup rate is possible if a large enough data set is processed. Due to memory limitations, larger datasets cannot be processed on our current facility. This section provided the details of these experiments.

3.4.1 Constraints

It is impossible to translate all types of Java code, even a *for*-loop, to a MapReduce function. For the prototype version of the J2M translator, many features cannot be realized. The constraints of the current translator are described as follows:

- Currently, the J2M translator can only work on Java *for* loops and only translates one loop each time.
- The target loop should be highlighted using J2M directives by users.
- The target loop should be located in a static method instead of an object method.
- No nested loop is available in the translator's current version.
- The target loop should be an independent loop.

- Users cannot declare an array inside the target loop. However, they can declare them elsewhere and use them inside the loop.

3.4.2 Sequential Java Program versus MapReduce Program

We use cloud servers instead of local machines for two reasons. The first reason is limitations in computing resources. For example, local computers may not have sufficient memory or the hard drive capacity to handle big data problems. Since users cannot load terabytes of data into computer memory, they must use a cloud computing technique even though a local computer might be faster than any single node of a server. Normally, if the data's size is not large enough, the programs executed on local computers can be faster than on cloud servers because the communication and preheating costs of cloud computing techniques may be expensive. Therefore, the experiments are designed to evaluate non-data-intensive cases. The second reason cloud servers are preferable is they can be faster than a local computer when executing compute-intensive programs. For example, if a user executes a program on a *N* nodes cloud server, it is *N* times faster than local computers in extreme cases. However, it cannot be processed due to the intercommunication cost.

This experiment measures the performance of the translation result in a huge number of floating point calculations, in which the total accumulated time is increased by five times and five billion for each time, on a five-node cloud server. The time curves of two programs are represented in Figure 11, where the cloud program is faster than the local program.

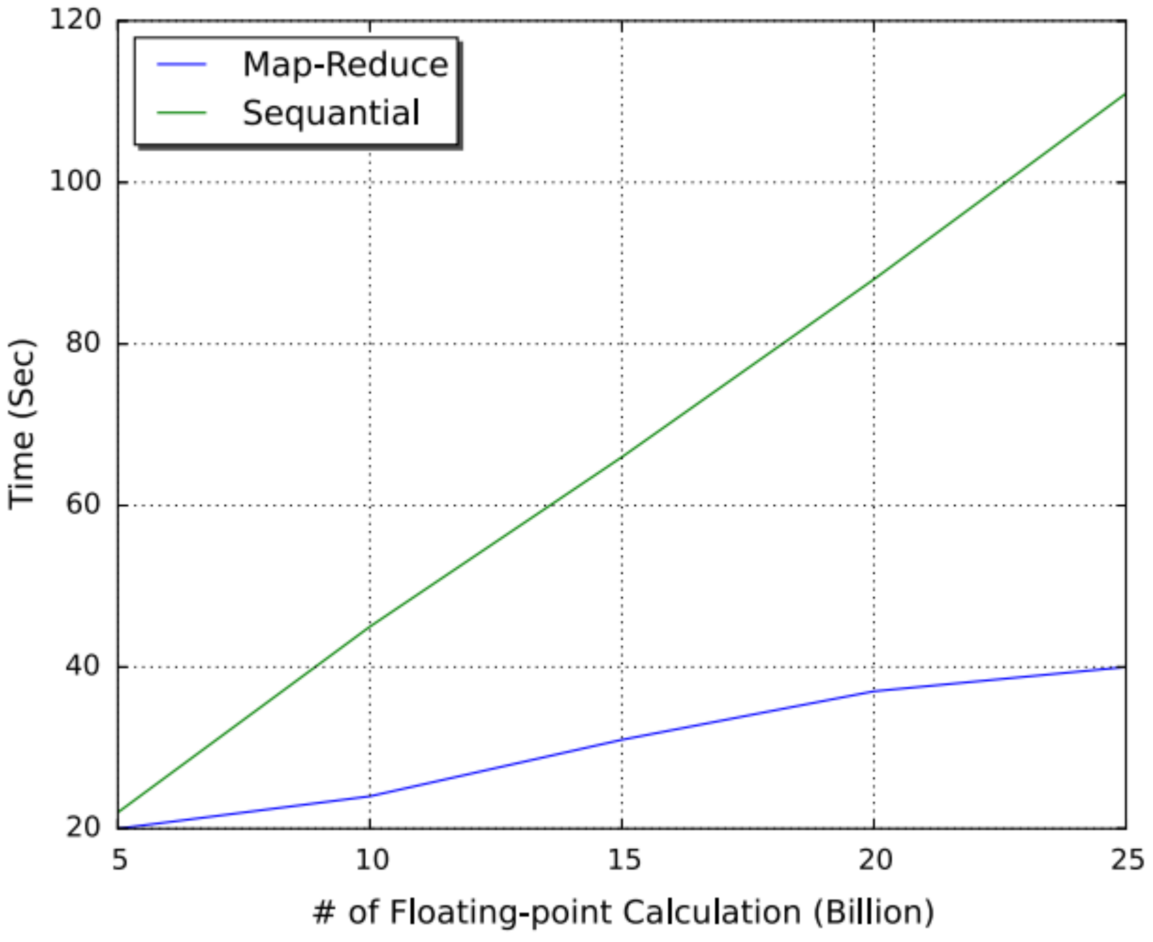

Figure 11 Performance result

Calculation times (Billion)	MapReduce execution time (s)	Local loop execution time(s)
5	20	22
10	24	45
15	31	66
20	37	88
25	40	111

Table 3 Test Result

Table 3 represents the time costs of five test cases. On a cloud server, the warming-up cost is the reason the time cost increment is less than the total time cost in the first test phase. To test the performance of cloud programs, the warming-up cost should be ignored because it varies for different servers. Therefore, the analysis should focus on the increment of time cost only.

The average time cost increment of the cloud program is 5.0 s while the average time cost increment of the local program is 22.25 s. Therefore, the time increment of the cloud program is 4.45 times faster than the local program, which is close to five times faster in extreme cases.

3.4.3 Performance on Different Calculation Sizes

When comparing a sequential Java program and a MapReduce program, the major disadvantage of a MapReduce algorithm, especially on a cloud server, is the MapReduce program spends more time on communication. It is why sequential programs are faster than parallel programs on small-size calculations. In the previous experiment, the MapReduce program was faster than the local program in all test cases because the calculation size was large enough. However, what is the performance of the translation results when different calculation sizes are involved, especially small-sized computations? The following experiment tests the performance on four different calculation sizes.

Before the experiment, a factor named speed up (SU) needs to be declared. It represents how the MapReduce program is faster than a sequential Java program. The formula for SU is as follows:

$$SU = \frac{Time\ Cost\ of\ Sequence\ Java\ Program}{Time\ Cost\ of\ MapReduce\ Program}$$

If SU is smaller than 1.0, the execution speed of the MapReduce program is slower than that of the sequential Java program. Otherwise, the MapReduce program's speed is faster.

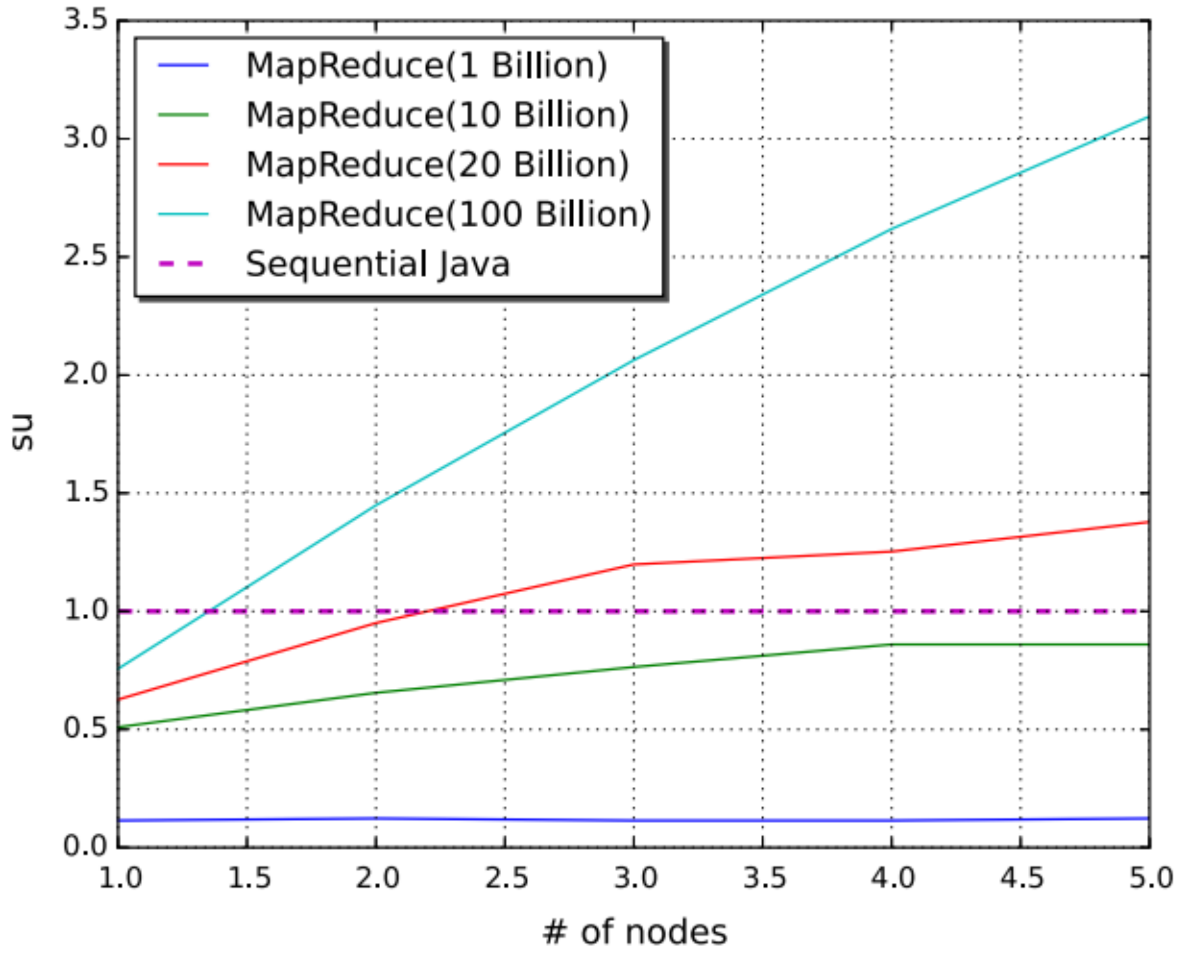

Figure 12 Performance on different calculation sizes

In this experiment, the test cases used four different calculation sizes: one billion, ten billion, twenty billion, and one hundred billion. The MapReduce program used one to five nodes on the cloud server in each of the five cases. The results are shown in Figure 12. The gray dashed line represents the sequential Java program, which is always 1.0. The blue and green lines are below 1.0, which means in these two test cases, the MapReduce programs are slower than sequential Java programs. The yellow line is above all the other lines. The red line is somewhere between the yellow and the green lines.

Compared to the blue line, the green line speeds up once the number of nodes increases. Conversely, there is no obvious change in the blue line. The reason is when the calculation size is small enough, the time cost of communication is equal to or greater than the speedup benefits in which the number of cloud server nodes increases. The other evidence is that the increments of red and green lines decrease to almost zero in the end. The increment can be negative when the number of cluster nodes is big enough.

The other problem is that all MapReduce programs are slower than sequential Java programs when there is one cloud server node. Besides the communication cost, there is a warming-up time cost. Cloud computing platforms like Hadoop need a certain amount of time to start up, initialize the environment, and load data. It is an variable factor that affects the execution time of cloud programs.

This experiment verifies that the performance of the translation result is better than the original sequential program when the calculation size is large enough. However, it is difficult to predict the exact size because it depends on the cloud server’s specific computing environment, such as CPU frequency, hard-disk access speed, and network transmission speed.

3.5 Conclusion

Cloud computing provides many benefits but also introduces many new challenges. One challenge is that one cannot run a Java code directly on a cloud platform effectively unless it is adapted to a cloud computing platform. In this paper, a translator for Java code with simple loops to MapReduce codes is proposed and implemented. It achieves automatic translation. Of course, the users should clearly understand their code to guarantee that there is no data dependency between loops. Our translator reduces the learning costs of users and hence also dramatically reduces the production cost. The experiments discussed in the previous section also demonstrate that the performance of the translated codes is accurate and efficient. In our current version, some features are not fully realized, and many restrictions are necessary for the translator to work correctly. In the future, more constraints can be overcome, and more loop cases will be studied. We hope that the proposed translator provides an initial step towards the automatic translation of traditional codes into cloud codes and a new framework in this exciting area.

4 JAVA TO SPARK TRANSLATION

Since the output program of the J2M translator is a MapReduce application running on a Hadoop platform, the source program's intermediate data must be uploaded to and downloaded from a Hadoop File System (HDFS). Therefore, the source program will be split into three stages: before loop, loop program, and after loop. Before the loop, the program prepares all necessary data and generates a text file that contains all data. Next, this file must be uploaded to the HDFS manually to be used for the loop program. The loop program is a MapReduce application that can be executed on the Hadoop platform, which will read data from the uploaded file and output result data to a new file on the HDFS. Afterward, users need to manually download the result file from the HDFS and execute the program after the loop that will read the result data from the downloaded file and continue the remaining tasks

This method is inconvenient because it requires developer participation. The HDFS operations also increase the time consumption of MapReduce programs, especially when the source program contains multiple loops that need to be translated. These problems are difficult to solve because, in the Hadoop framework, all applications communicate through HDFS [40]. If the intermediate data is large enough, the performance of the translation result may be worse than the source program. To avoid this problem, executing the resulting programs on an in-memory cloud computing platform is a possible solution.

A Java to Spark (J2S) translator is proposed in this chapter; it is built on the J2M translator's framework. [41] Compared to the J2M translator, the J2S translator generates a Spark program instead of a Hadoop program, and it can translate three types of source code: for-loop, task, and hybrid loop and task. Benefiting from Spark's in-memory model [42], the J2S translator can communicate with a Spark platform directly without generating intermediate files. Furthermore, since no file operations are required in the translation procedures, the resulting program is not split into three pieces, and no human participation is required while the program is running.

However, the in-memory model requires more memory space than the HDFS model. Therefore, the translation capability of the J2S depends on the working environment's memory capacity, the computing cluster where the resulting program is deployed. Specifically, this system does not work if the intermediate data size is larger than the memory capacity. Meanwhile, the in-memory model also increases hardware costs because large memory systems are more expensive. [43] All of these are trade-offs of implementing the in-memory model. Developers should analyze their use cases before using the J2S translator.

The primary technical challenges in code translators are tokenization, syntax analysis, and the resulting program template design and code generation unique to Spark. Since the J2S translator is the successor of the J2M translator, tokenization, and syntax analysis have been solved in the current version. Therefore, the major challenge for the J2S translator is determining how to design the templates and generate codes to implement more functionalities and perform more efficiently. The following sections demonstrate the details of template design for three use cases.

The following sections describe the design and evaluation of for-loop parallelization, task parallelization, and for-loop-in-task parallelization, respectively. The last section discusses future work and the conclusion of the J2S project.

4.1 For-Loop Parallelization

Like the Headbutt platform, Spark also provides a map function. The translation mechanism of J2M can also be applied to Java for-loop to Spark code translation. The difference between the two translation types is the in-memory working mode of Spark programs. Therefore, the Spark program always contains all three process stages (i.e. data preparation, processing, and output) in one program instead of generating three individual programs.

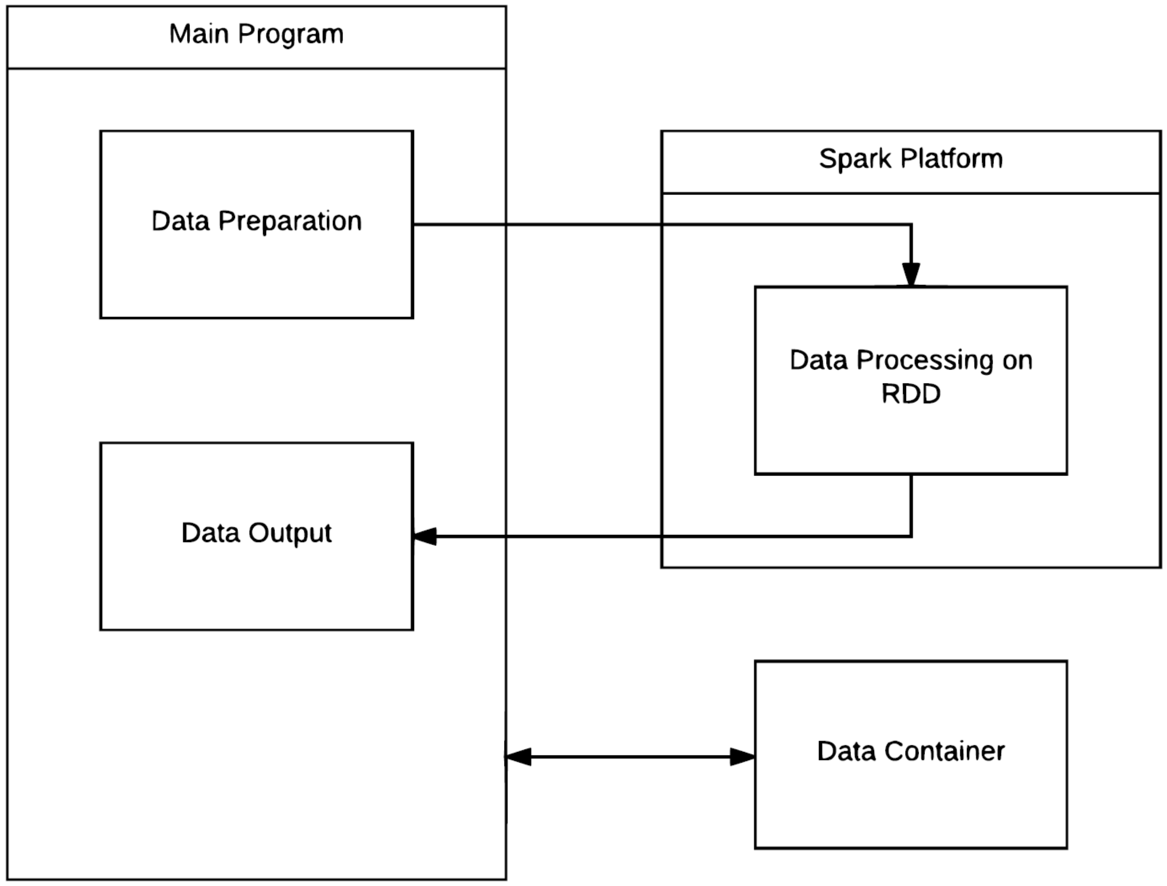

Figure 13 The structure of the for-loop translation resulting program

Figure 13 describes the program structure of a translation resulting program containing two major parts, which are the main program and a data container. All three data processing stages are performed in the main program. In the data preparation stage, all intermediate data are packaged and submitted to Spark resilient distributed datasets (RDD) [44]. In the next two steps, the data are processed on the Spark platform, and the output is collected from Spark RDD. In addition to the main program, a data container class is designed to contain all intermediate data for communication between local memory and the Spark RDD.

4.1.1 Data Container

In Java to Spark translation, one data container class is designed to contain all intermediate data when communicating with Spark RDD. Spark RDD is a distributed data container, and each RDD entity can hold only one data object. However, intermediate data may have a significant number of variables, so these variables need to be packaged into one object.

```
    public void test(int time) {
        for (int i = 0; i < 10; i++) {
            num[i] = i;
        }

        //start
        for (int i = 0; i < 10; i++) {
            num[i] += calculation(i,time);
        }
        //end

        for (int i = 0; i < 10; i++) {
            System.out.println(num[i]);
        }
    }
```

Code 1 A Source Program Example

```
class LoopData implements Serializable{
    public int i = 0;
    public int num_0 = 0;
    public int time = 0;

}
```

Code 2 A Data Container Example

Code 1 is an example of a Java program that can be translated by the J2S translator. The for-loop in this program, which is marked by two directives *start* and *end*, is being translated. The Java class *LoopData* in Code 2 is the data container in the translation resulting program. The class will be used to create data objects that are later inserted into one Spark RDD container.

There are three variables in the targeted for-loop: *i*, *num[i]*, and *time*. Three corresponding variables, *i*, *num_0*, and *time*, are declared in the *LoopData* class to handle all variables in the targeted for-loop. The *num[i]* variable is one element in the *num* array. Since only one element is processed in each iteration of the for-loop, it is not necessary to transmit the entire array into each Spark RDD element. Thus, the *num_0* variable is declared to replace the *num[i]* variable in the data container class.

4.1.2 Data Preparation

```
//RDD initialization
List<LoopData> ld = new LinkedList<LoopData>();

for (int i = 0; i < 10; i++) {
    LoopData lde = new LoopData();
    lde.i = i;
    lde.num_0 = num[i];
    lde.time = time;
    ld.add(lde);
}

//Spark setup
SparkConf conf = new SparkConf().setAppName("Template");
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<LoopData> loopData = sc.parallelize(ld).cache();
```

Code 3 A Data Preparation Program Example

Data Preparation is the first step of processing data on Spark platform. The portion of code in Code 3 is the data preparation stage translated from the example Java program in Code 1. RDD initialization and Spark configuration occur during this stage.

A data container list is declared during RDD initialization. In a data container, the values of variables in each loop iteration are assigned to the corresponding variables in the original java program. The loop iterator can be useless if it only controls the number of iterations but does not participate in data processing. However, to simplify translation, the proposed translator is not designed to detect whether the iterator is useless, and it always transmits the loop iterator variable to Spark RDD.

The Spark configuration part is responsible for initializing the Spark platform and transmitting the data container list to the Spark RDD system. If the source Java program contains more than one for-loops that need to be translated, the Spark configuration code should be at the top of all targeted for-loops, and it is shared among all Spark tasks rather than configured separately.

4.1.3 Data Processing

```
JavaRDD<LoopData> sparkResult = loopData.map(new Function<LoopData, LoopData>() {
    public LoopData call(LoopData loopData) throws Exception {
        //loop data
        int i = loopData.i;
        int num_0 = loopData.num_0;
        int time = loopData.time;
        //loop
        num_0 += calculation(i,time);
        //output
        LoopData lde = new LoopData();
        lde.i = i;
        lde.num_0 = num_0;
        lde.time = time;
        return lde;
}});
```

Code 4 A Data Process Program Example

The data processing stage is the main procedure in the translation resulting program. Code 4 is an example of the data process stage in the translation result. The Spark map function in this stage is implemented by migrating instructions in the original for-loop. It is then processed on the Spark cluster. Specifically, the original for-loop is implemented as distributed processes in this stage.

There are three steps in the map function: data loading, processing, and storing. In the first step, data loading, data are loaded from the Spark RDD, and all variables are stored in the data container. Temporary variables *i*, *num_0*, and *time* are created to hold the corresponding values. Next, these data are processed in a statement that is migrated from the original for-loop. This migration is no more than a simple copy-and-paste, leaving array variables as the only variable needing modification, such as the *num_0* variable in Code 4. Since only partial elements of an array are required to be processed in an iteration, they are converted to non-array variables.

Finally, the data in the Spark RDD must be updated. This step simply assigns the values of all temporary variables to the data container, which is the opposite of the loading step in the Spark map function. To simplify this procedure, all variables are updated without a modification check. Regardless of whether the variable is modified, the update procedure is always carried out.

4.1.4 Data Output

```
ld = sparkResult.collect();
for (int i = 0; i < 10; i++) {
    LoopData lde = ld.get(i);
    i = lde.i;
    num[i] = lde.num_0;
    time = lde.time;
}
```

Code 5 An Output Program Example

Data Output is the last stage in the resulting program, and it is the opposite of the data preparation stage. It first collects the processing result from the Spark RDD and then updates all values in local memory. Since the loop iterator is assigned inside of the loop, it is crucial to ensure that all variables in the updated procedure remain in the same order when they are uploaded.

4.1.5 Evaluation

To evaluate the performance of the translation results, all tests are executed on a computer cluster consisting of five computing nodes with an Intel Xeon E5-2650 CPU and 64GB of memory. In this experiment, a sequential Java for-loop processes 100 billion floating-point calculations. The Spark program, translated from Java code using the proposed translator, is executed five times with a varying number of cores. The following table and figure describe the results of this experiment.

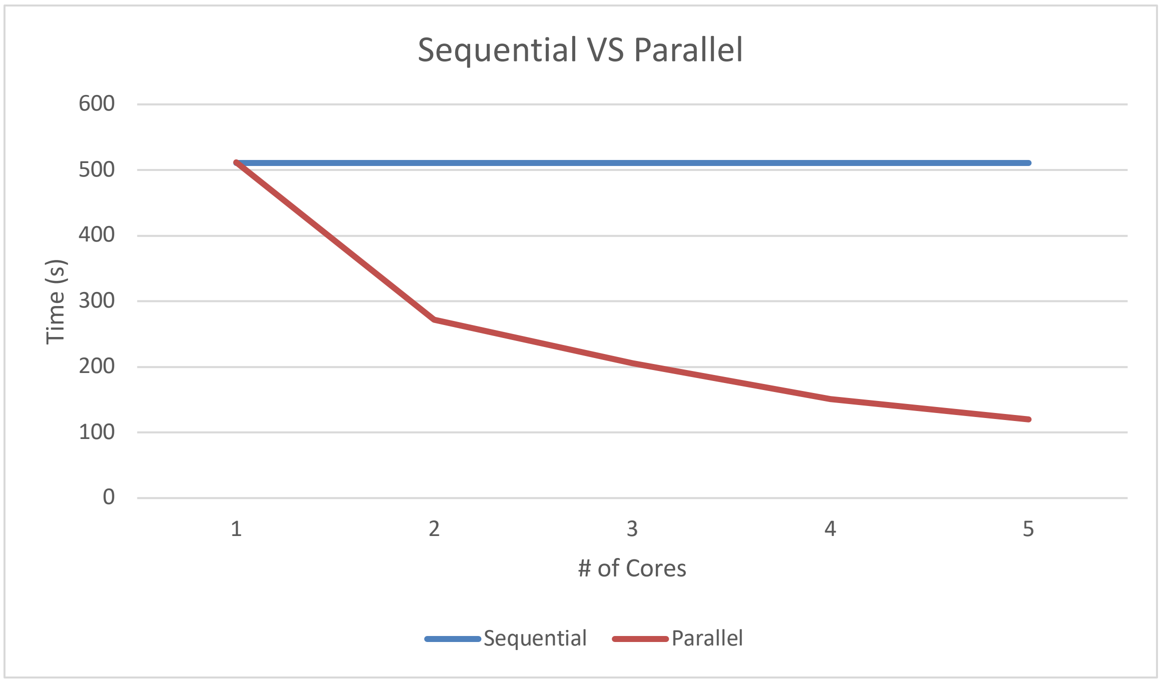

Figure 14 Time consumption versus the number of cores

Core #	1	2	3	4	5
Sequential(s)	510.725	-	-	-	-
Parallel (s)	511.727	271.76	205.502	150.872	119.921
Speed-Up	0.998	1.879	2.485	3.385	4.259

Table 4 Time Consumption Versus the Number of Cores

Figure 14 shows that the time cost of the parallel program decreases as the number of cores increases. The last row in Table 4 demonstrates the speedup rate of the Spark program, whose values are calculated by dividing the time cost of the sequential program over that of the parallel program. Theoretically, the optimal speedup rate should be equal to the number of computing cores used in the experiment. However, in practice, it is impossible to reach the maximum value due to overhead costs. The speedup values in Table 4 are quite reasonable because they are close to the optimal values.

4.2 Task Parallelization

```
public void process() {
    int a = 10000, b = 10000;      (1)
    a = f1(a);                     (2)
    b = f2(b);                     (3)
    int c = a + b;                 (4)
}
```

Code 6 A Source Program Example

Another problem is multi-task parallelization. A simple example is displayed in Code 6. Two variables, a and b, store the returned values from two functions and they are then accessed in the following statement. These two variables are independent and will not be used until statement 4. Therefore, before the program accesses the two variables, these two instructions can be processed in parallel.

Since it is processed on the Spark platform, this problem can be more complicated than the loop-parallelization problem because the platform is not designed for this purpose. Two problems should be solved in this type of translation. The first problem is converting the synchronous program to asynchronous, and the second problem is processing the task on the Spark platform.

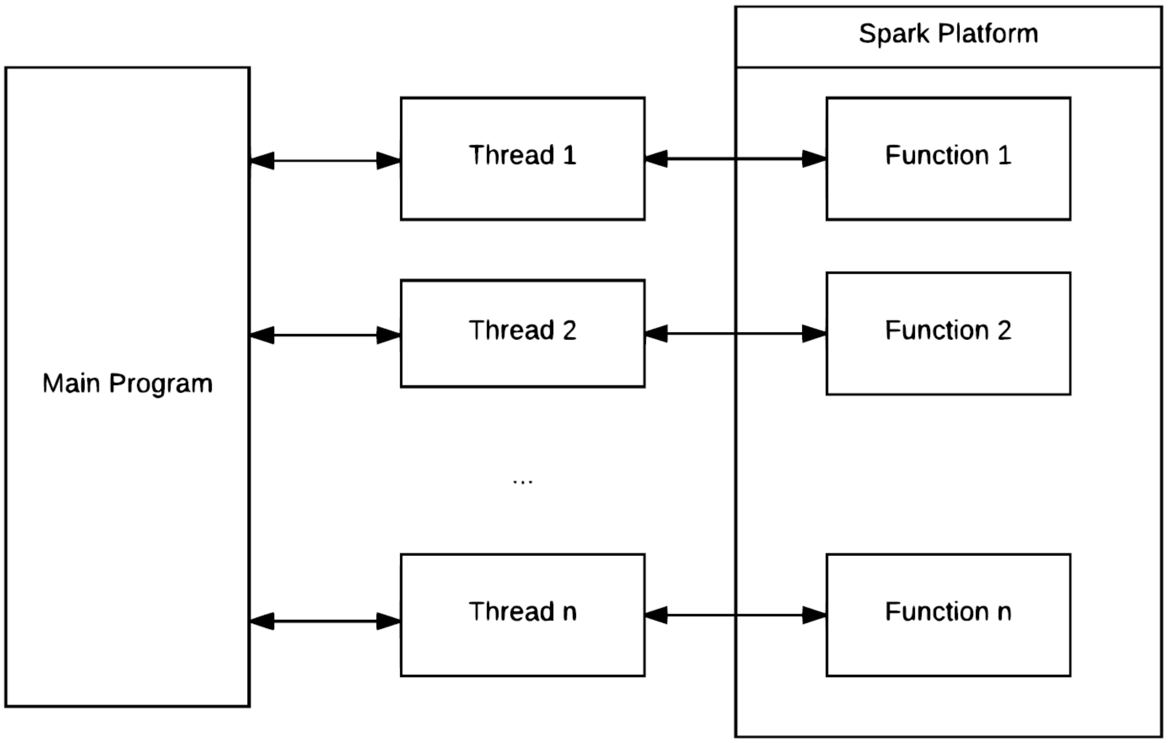

Figure 15 The structure of the translation resulting program

The proposed design of this type of translation is represented in Figure 15. Compared with the loop-parallelization problem, the task-parallelization problem has an additional intermediate layer between the main program and the Spark platform. The targeted instructions are processed in individual threads, and then the Spark tasks are implemented in each thread. Each thread controls only one Spark task; thus, they can be processed concurrently.

4.2.1 Synchronous to Asynchronous

In the synchronous mode, the execution of an instruction should wait until the previous instruction is terminated in a regular Java program. Although this mode is easily controlled and understood, there is a problem. For instance, in Code 6, statement 3 will wait until the end of statement 2, although these two instructions can be processed at the same time due to their independence.

In the asynchronous mode, the original function can be processed in individual threads. It is only blocked by statements like statement 4 in Code 6. This instruction needs to access variables *a* and *b*, so it should wait until the results of statements 2 and 3 are returned. Statements 2 and 3 can be processed under the asynchronous mode, and no blocker is required until statement 4.

```
int a = 10000, b = 10000;
//thread configuration
ExecutorService executorService = Executors.newFixedThreadPool(2);

//group  1
//statement 2
final int a_tmp = a;
Future<Integer> future1 = executorService.submit(()->f1(a_tmp));

//statement 3
final int b_tmp = b;
Future<Integer> future2 = executorService.submit(()->f2(b_tmp));


//group 2
//statement 2 locker
try {a=future1.get().intValue();}
catch (Exception e) {e.printStackTrace();}

//statement 3 locker
try {b=future2.get().intValue();}
catch (Exception e) {e.printStackTrace();}

//statement 4
int c = a + b;

//thread  configuration
executorService.shutdown();
```

Code 7 A Translation Resulting Program Example

The code in Code 7 is the translation result of the program in Code 6. Each source instruction in Code 7 is translated into three statements. These statements are divided into two groups and located in two different places. One group is at the original location in the source code, and the other is placed before statement 4 in the translation result.

The first group has two types of statements, the temporary variable declaration, and the thread statement. In the proposed design, the Java executor service function is used to handle multi-thread operations. Since all variables in the executor service function should be constant, temporary variables are required to store them. Afterward, the original function is simply placed in a new thread with the constant parameters.

The second group is the thread locker, which is in the previous line before the instruction for accessing the result values, such as statement 4 in Code 7. If this instruction requests more than one variable, all thread lockers should be placed before it. For example, in Code 7, both lockers are on the top of statement 4.

In addition to these statements, a thread configuration and a thread closure are placed at the top and the bottom of this function, respectively. To maximize performance, the thread pool's size is set to be the same as the demanded number of targeted instructions.

```
public void process() {
    int a = 10000, b = 10000;

    //j2s_task 1
    a = f1(a);

    //j2s_task 2
    b = f2(b);

    //j2s_locker 1
    //j2s_locker 2
    int c = a + b;
}
```

Code 8 A Directive Example

Like the loop-parallelization function, directives are also required to mark the targeted instructions and the locations of the blockers. Users are responsible for determining which instructions are to be translated and the places to insert blockers. However, two types of directives are used in the task-parallelization function, which are *j2s_task* and *j2s_locker*. A number is listed after these keywords to identify the threads. It should be noted that these two types of directives work in pairs. In the proposed design, the size of the thread pool is set automatically according to the number of targeted instructions.

4.2.2 Process Single Function on Spark

The second step in task-parallelization translation is processing these instructions on the Spark platform. The major problem in this step is the Spark platform is not designed for single function processing. Generally, the RDD contains a large amount of data, and it processes them in parallel; in this specific task, there is only one function to be processed.

The proposed solution is to create a small RDD that contains only one pseudo data and then uses it to load a single function task. The following code piece is an example of the translation result.

```
Future<Integer> future1 = executorService.submit(()->{
    List<String> rdd_data = Arrays.asList("a");
    JavaRDD<String> rdd = sc.parallelize(rdd_data);
    return rdd.map(s->f1(a_tmp)).collect().get(0);
});
```

Code 9 An Asynchronous Spark Program Example

The statements in Code 9 describe how to process a single function on the Spark platform. It replaces the thread instructions in Code 7. This function creates an array consisting only one string, "a.” This is pseudo data; it is only used to create an entry for the Spark application and can be changed to any other value. The following statements in this function convert the pseudo data array to RDD, and then they launch the Spark function. Each targeted instruction leads to an individual Spark task so the original functions can be processed on the Spark platform.

4.2.3 Evaluation

To evaluate the performance of the task-parallelization translation, the next experiment uses the same computer cluster as the one in the loop-parallelization chapter. Unlike the previous one, in the Task-parallelization translation, only a few functions need to be parallelized because in normal cases, a small number of asynchronous functions is adequate for a single program. Since the program does not benefit from increasing the number of computing cores, the configuration of Spark is fixed to 5 execution cores.

Instead of manipulating the number of computing cores, this experiment focuses on the number of calculations. There are five asynchronous functions, and each performs a different number of floating-point calculations from 1 billion to 5 billion. Table 5 and Figure 16 describe the time consumption of all tests.

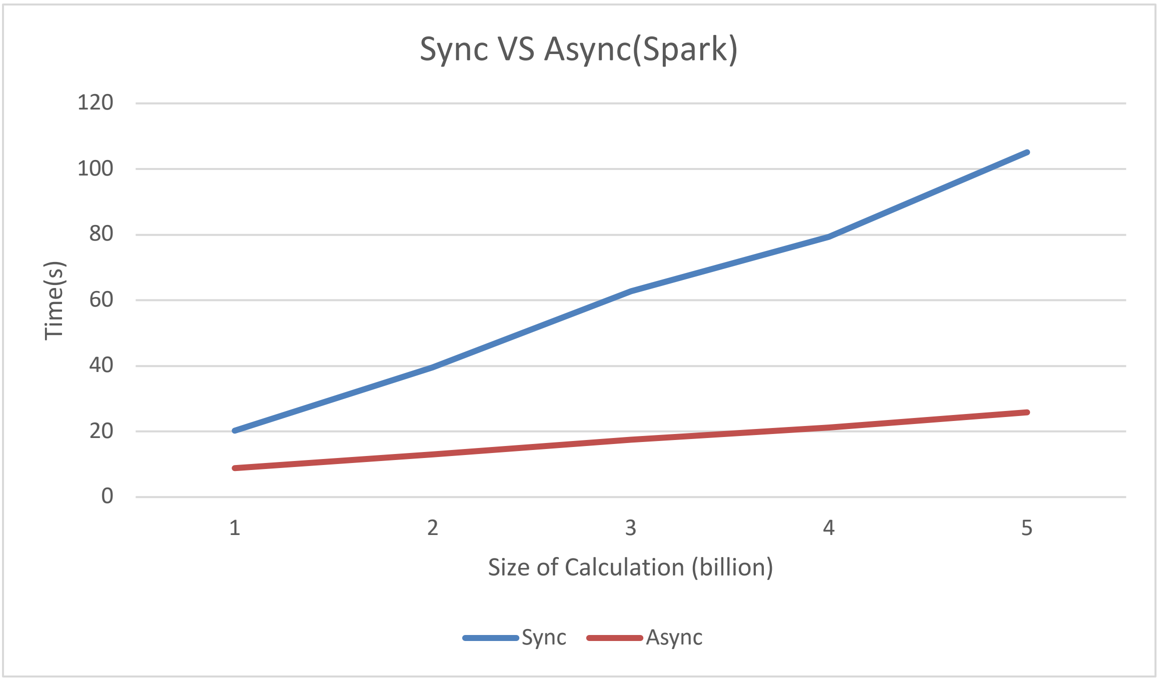

Figure 16 Time consumption versus calculation size

Size of Calculation (billion)	**1**	**2**	**3**	**4**	**5**
Synchronous(s)	20.239	39.557	62.828	79.248	105.112
Asynchronous and Spark (s)	8.822	12.977	17.438	21.282	25.833
Speed-Up	2.294	3.048	3.603	3.724	4.069

Table 5 Time Consumption Versus Calculation Size

In Figure 16, the blue and orange lines represent the execution times of the synchronous function and the asynchronous Spark function, respectively. The blue line has a much greater growth rate, which indicates that the Spark functions benefit more from the larger data size.

Compared with the loop-parallelization translation, this type of translation has more overhead costs because it adds one more logical layer. The two overhead costs, one from the Java thread pool and the other from the Spark platform, are almost constant if the size of the thread is fixed. In conclusion, the rate at which the speed increases can be greater with larger datasets.

4.3 For-Loop in Task Parallelization

Generally, functions with iteration or recursion consume most of the resources in a single program. Also, major cloud computing platforms are typically not designed for recursion, and parallelization is difficult due to data dependency. However, iteration is much easier to parallelize if there is no data dependency, such as in loop-parallelization translation.

During the translation research, a new question arose: can the loop-parallelization and task-parallelization problems be combined? If asynchronous functions in the task-parallelization translation contain larger iterations, can the time consumption be reduced by parallelizing these iterations? To solve this problem, a combination translation is discussed in this chapter.

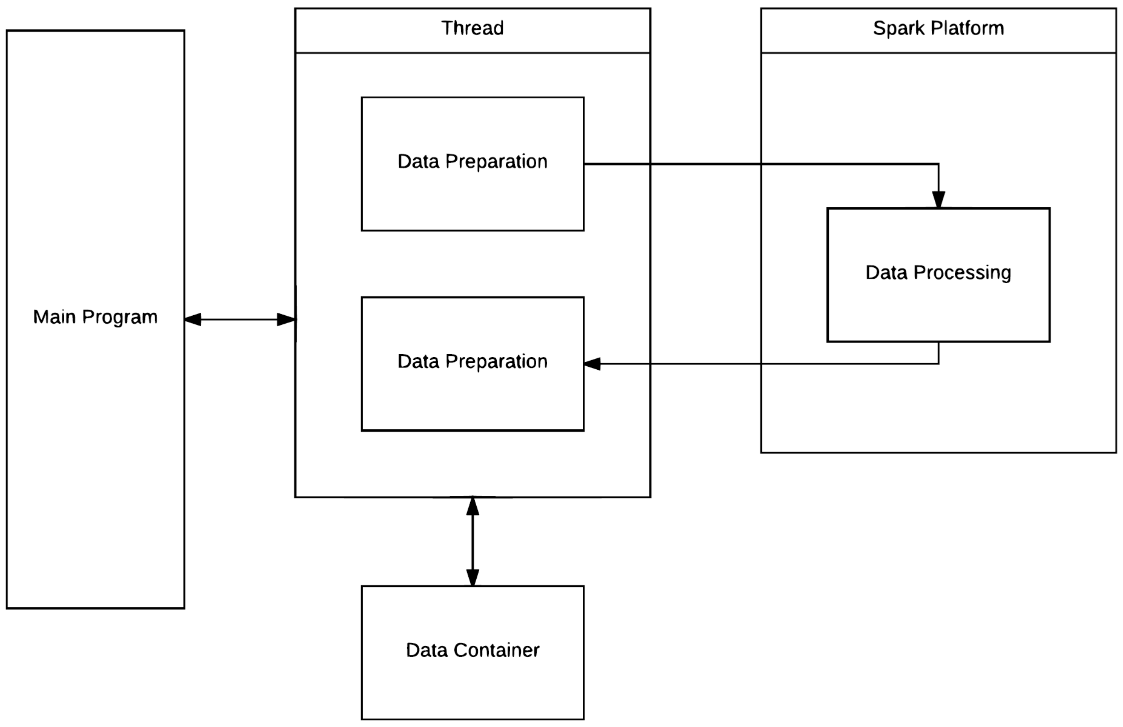

Figure 17 The Structure of Translation Resulting Program

Figure 17 describes the design of the loop-in-task translation. In short, this type of translation combines the two previous types. Loop-parallelization statements are included in the asynchronous functions of task-parallelization translation. It is similar to implementing the loop-parallelization translation method in each function and parallelizing the functions in the asynchronous mode. Therefore, compared with task-parallelization translation, in the new design, for-loops are processed on the Spark platform instead of the asynchronous functions.

Since the Spark platform does not allow an RDD function to be embedded within another function, only the for-loop part of the asynchronous function, not the entire function, can be processed on the Spark platform. In fact, as mentioned previously, this type of translation focuses on the functions that consume the most resources during iteration. Processing for-loops in parallel on the Spark platform can significantly reduce the time cost of such a program. If the program being translated does not contain a large loop, using the task-parallelization method is a more efficient solution.

```
public int f1(int n) {
    …
    for (…) {…}
    …
}
public int f2(int n) {
    …
    for (…) {…}
    …
}
public void process() {
    int a = 10000, b = 10000;
    a = f1(a);
    b = f2(b);
    int c = a + b;
    System.out.println(c);
}
```

Code 10 A Source Program Example

Code 10 describes an example source code of this type of translation. Its main function is the same as that in the Task-parallelization translation example shown in Code 6. However, each of the *f1* and *f2* functions in this example contains a large for-loop. Inside of the main function, the *f1* and *f2* are called and must be processed in asynchronous mode.

Two parts in this program need to be translated in this type of translation. First, in the main thread, translated from the synchronous mode to the asynchronous mode, the targeted instructions generate the result codes, which are the same as the codes described in the first step of the task-parallelization translation in Code 7. The second part is the side functions, which are *f1* and *f2* in this example. Their result codes are similar to the codes in the loop-parallelization translation. However, the Spark configuration should be moved to the main function of this program because all side functions share the same configuration.

```
public void process(int n) {
    …
    SparkConf conf = new SparkConf().setAppName("Test");
    JavaSparkContext sc = new JavaSparkContext(conf);
    ExecutorService executorService = Executors.newFixedThreadPool(5);

    final int a_tmp = a;
    Future<Integer> future1 = executorService.submit(()->func(a_tmp,sc));
    …
}
```

Code 11 A Spark Configuration Example

When Spark configuration statements are inserted, these statements need to be placed inside of the main function. Otherwise, the Spark cluster cannot initialize the Spark tasks. The proposed solution declares the configuration statements in the main function, and it adds a new reference argument in the functions containing the target for-loops, such as the example described in Code 11. A new reference parameter *sc* is added to the *func* function's reference.

Container classes are also required for the translation of loop-parallelization. Each for-loop needs a specific container class to hold all loop data. Furthermore, if this program contains *N* targeted loops in total, the resulting program should also have *N* corresponding container classes.

4.3.1 Evaluation

The evaluation method for the loop-in-task translation is more complicated than that of the previous two chapters. The performance of the program resulting from the translation with two configurable variables, the number of cores and the size of the floating-point calculation, should be tested. The translation method used in this chapter is the combination of the two previous approaches. This experiment is also implemented on the same computer cluster as the previous experiments.

Size of Calculation (billion)	1	2	3	4	5
Sequential(s)	19.18	38.129	57.258	76.387	95.856
Spark (s)	8.9	12.627	16.887	21.736	25.548
Speed-Up	2.155	3.02	3.39	3.514	3.752

Table 6 Time Consumption versus Calculation Size

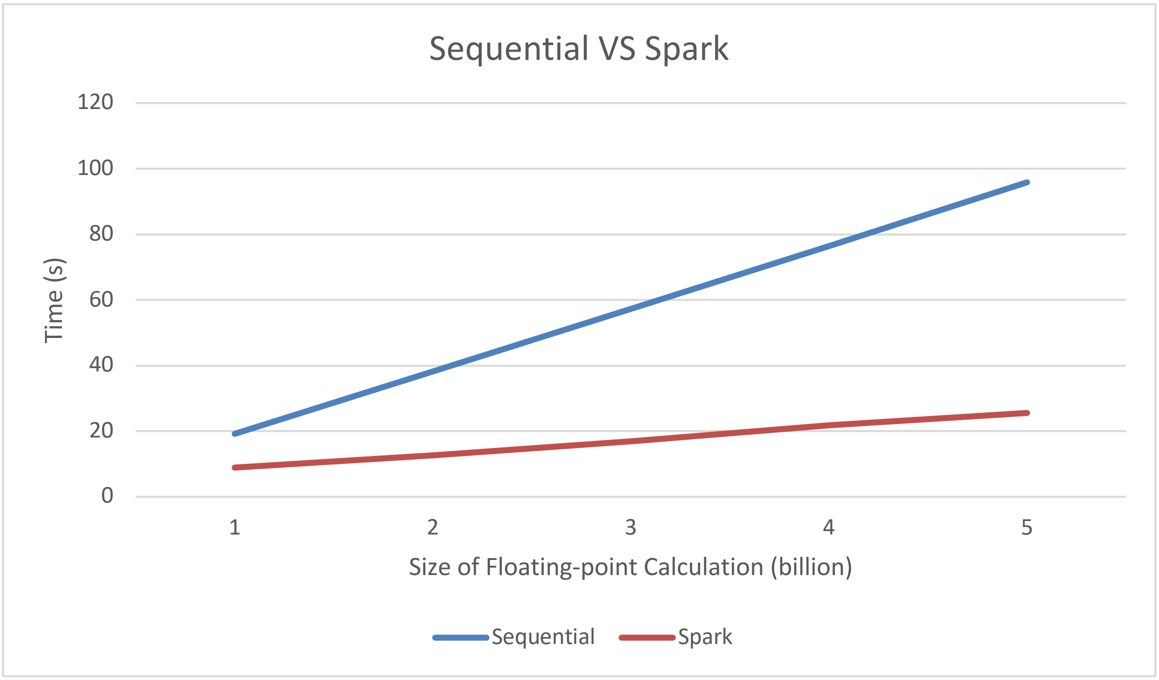

Figure 18 Time consumption versus calculation size

The first experiment compares the performance of the original program and the resulting program on different calculation sizes. The experiment's results are described in Figure 18 and Table 6. This test is similar to the one in the task-parallelization chapter. However, as shown in

Table 6, the speedup increment is smaller than in the task-parallelization chapter because the overhead cost of the loop-in-task translation is greater.

The maximum speedup in this configuration is five, which is the same as the number of cores. Also, the speedup rate will get closer to five as the size of the calculation grows. This conclusion is based on a comparison of results from the first test in Table 7 and all the results in Table 6. The speedup in the first test in Table 7 is larger than all of those in Table 6 when the calculation size in the former is 10 billion, which was the largest tested.

Cores #	**1**	**2**	**3**	**4**	**5**
Sequential(s)	190.903	-	-	-	-
Spark (s)	46.195	24.496	19.881	17.041	13.351
Speed-Up	4.133	7.793	9.6	11.202	14.299

Table 7 Time Consumption Versus the Number of Cores

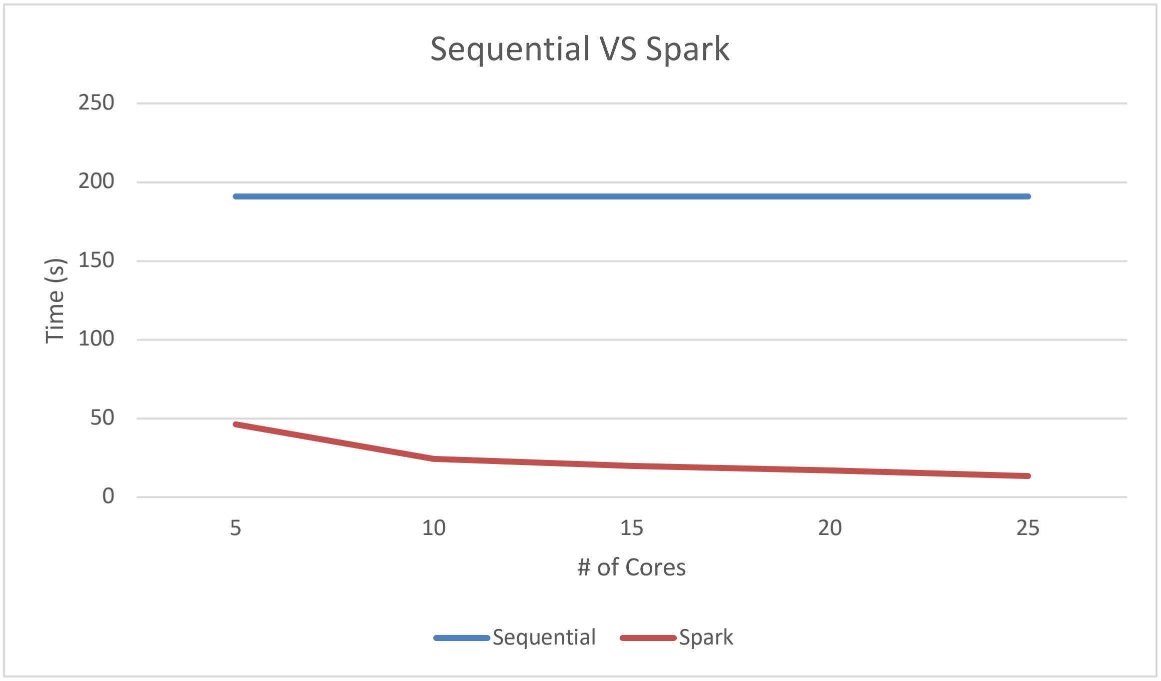

Figure 19 Time consumption versus the number of cores

The second test is to fix the calculation size to 10 billion and manipulate the number of computing cores. Figure 19 shows that the Spark program benefits from the increased number of cores in all five tests. Meanwhile, the speedup increment decreases as the number of cores increases, as shown in Table 7, because the calculation size is not large enough. When processing time is limited, the overhead cost takes more proportion. Since the overhead cost is practically fixed, it will use a greater proportion of the total time consumption when the latter is lower. Furthermore, the increment of speedups can be negative if the data size is small enough. In this case, the time cost will increase if the number of cores keeps increasing.

4.3.2 Conclusion

The J2S translator is a new generation of the J2M translator. These two translators have the same framework design, but the J2S translator has more functionality such as the ability to translate more types of source programs and is built on the in-memory model. Evaluations with the three types of translation demonstrate that all of three translation results work well in their domains. Meanwhile, the J2S translator is more productive because it doesn't require human participation while the procedure is running.

In the current version, the J2S translator still has some disadvantages. For example, it packages all intermediate data, including useless ones, and sends them to the cloud platform. It will also transmit redundant data. The performance of the resulting program can be significantly improved if these problems are solved. Also, the J2S translator can only translate Java programs currently. However, besides Java, there are many popular programming languages for processing big data. Implementing the automatic translation of code written in these languages can create powerful and efficient tools for data engineers. The J2S translator only provides preliminary ideas for automatic translation from sequential to parallel programs on a cloud platform, and we hope related projects in the future will gain inspiration from ours.

5 SEMI-AUTOMATIC TRANSLATION

The two previous chapters demonstrate two Java code translators that can automatically translate sequential Java programs to parallel cloud computing applications. They are simple and powerful, especially the Java to Spark translator. Due to Spark's in-memory working mode, human participation is not required when loading and unloading data. These two chapters also provide some proof that these two translators work well in their designed working domains, but how well do they perform in other scenarios?

When a data fragment is very small (e.g., the size of an integer number or even a single character), the overhead cost of communication between each computing unit is sometimes larger than the benefits of the parallel processing model. This problem is more serious on the cloud computing platform. However, when processing these data in a sequential loop normally, the overhead is minimal. Therefore, when processing tiny data fragments, using loops is a common method.

Is there any data dependency in this coding pattern? Not necessarily, it depends on other conditions. Thus, if there is no data-dependency issue, logically, the program could be translated by automatic translators. The problem is the performance could be very poor. The new question is how the translators can be used to translate this type of source code.

Semi-automatic translation is the solution proposed to handle this type of translation. The translators can translate loops but cannot control the data size. Furthermore, if developers do not want to deal with developing a cloud computing application but still need to deploy their program on a cloud platform, they can handle the data-size problem by themselves and then use the translator to migrate their program to a cloud platform. This situation involves semi-automatic translation. This chapter uses an example, the translation of a tensor multiplication program, to demonstrate semi-automatic translation.

5.1 Tensor

Tensor is a geometric term representing linear relationships between geometric objects such as a scalar or a vector. [45] It is the superset of a scalar, vector, and matrix. A scalar is a 0th-order tensor, a vector is a 1st-order tensor, and a matrix is a 2nd-order tensor. There are 3rd-, 4th-, and even nth-order tensors. [46] The complexity of the tensor increases as the order gets larger.

$$\mathrm{S}$$

Equation 9 A Scalar, 0th-order Tensor

$$|V_1\ V_2\ \dots\ V_x|$$

Equation 10 A Vector, 1st-order Tensor

$$\begin{vmatrix} M_{1,1} & M_{1,2} & \dots & M_{1,x} \\ M_{2,1} & M_{2,2} & \dots & M_{2,x} \\ \vdots & \vdots & \ddots & \vdots \\ M_{y,1} & M_{y,2} & \dots & M_{y,x} \end{vmatrix}$$

Equation 11 A Matrix, 2nd-order Tensor

$$\begin{vmatrix} T_{1,1,1} & T_{1,1,2} & \dots & T_{1,1,x} & T_{2,1,1} & T_{2,1,2} & \dots & T_{2,1,x} & & T_{z,1,1} & T_{z,1,2} & \dots & T_{z,1,x} \\ T_{1,2,1} & T_{1,2,2} & \dots & T_{1,2,x} & T_{2,2,1} & T_{2,2,2} & \dots & T_{2,2,x} & & T_{z,2,1} & T_{z,2,2} & \dots & T_{z,2,x} \\ \vdots & \vdots & \ddots & \vdots\ , & \vdots & \vdots & \ddots & \vdots & ,\dots, & \vdots & \vdots & \ddots & \vdots \\ T_{1,y,1} & T_{1,y,2} & \dots & T_{1,y,x} & T_{2,y,1} & T_{2,y,2} & \dots & T_{2,y,x} & & T_{z,y,1} & T_{z,y,2} & \dots & T_{z,y,x} \end{vmatrix}$$

Equation 12 A 3rd-order Tensor

Tensors are widely used in many scientific areas such as mathematics, physics, and computer science. Many mathematic operations can be applied to tensors, such as a cross product and a dot product. Depending on the size and order, the time complexity of these mathematic calculations can be very large. Thus, improving the performance of tensor operations is fundamental research and can significantly reduce the processing time of many mathematic applications. A 3rd-order tensor multiplies by a matrix and is selected as an example of semi-automatic application translation.

5.2 Tensor Multiplication

A 3rd-order tensor times a matrix is similar to a vector times a scalar. In the vector multiplication, the scalar will be multiplied with each element in the vector respectively [47]. The resulting vector is the original vector scaled by the scalar.

$$s \times \begin{bmatrix} v_1 \\ v_2 \\ \cdots \\ v_n \end{bmatrix} = \begin{bmatrix} sv_1 \\ sv_2 \\ \cdots \\ sv_n \end{bmatrix}$$

Equation 13 Vector multiply Scalar

In the multiplication between the 3rd-order tensor and matrix, the 3rd-order tensor is a vector of matrices, and each element in this vector is a matrix [48]. The matrix in this calculation is also a scalar. Therefore, the multiplication can be split into two steps. The first step is applying the matrix to each element in the 3rd-order tensor. The second step is calculating the matrix multiplication.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,1} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,p} \\ b_{2,1} & b_{2,1} & \cdots & b_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,p} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,p} \\ c_{2,1} & c_{2,1} & \cdots & c_{2,p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,p} \end{bmatrix}$$

$$c_{i,j} = \sum_{k=1}^{m} a_{ik} b_{kj} \text{ , (i = 1…n, j = 1…p)}$$

Equation 14 Matrix Multiplication

For example, a 3rd-order tensor has *s* matrices, and each matrix has *n* rows and *m* columns. Let this tensor multiply a matrix that has *m* rows and *p* columns. The result is a new 3rd-order tensor that has *s* matrices according to Equation 13, and each matrix has *n* rows and *p* columns according to Equation 14.

5.3 Program Design

```
for(int i = 0; i < s; i++){
    for(int j = 0; j < n; j++){
        for(int k = 0; k < p; k++){
            result[i][j][k] = 0;
            for(int l = 0; l < m; l++){
                result[i][j][k] += tensor[i][j][l] * matrix[l][k];
            }
        }
    }
}
```

Code 12 Tensor Multiplication

Code 12 is a simple Java program that processes the tensor multiplication example stated at the end of the previous section. The code is short but complicated because it has four for-loops. The time complexity is not $o(n^4)$ and therefore difficult to determine because of the loops. Since *s, n, p,* and *m* are varied and may not be the same, the accurate time complexity is $o(\mathrm{s} * \mathrm{p} * \mathrm{n} * \mathrm{m})$.

This program can be translated by our translators because each iteration of the outer three for-loops is independent. However, the resulting program's performance may be very poor if it is translated directly because the partition in the Spark RDD is very small. This program also contains four nested loops which make the translation more complicated and unstable. To solve the problems stated above, a manual operation on this program is required to resize the loop and simplify the code.

```
//Step one
tensor_fragment = split_tensor(tensor);

//Step two
for (int i = 0; i < tensor_fragment.length; i++) {
    result[i] = tensor_multiplication(tensor_fragment[i], matrix);
}

//Step Three
tensor = combine_result(result);
```

```
public static Tensor[] split_tensor(Tensor tensor){

    //number of fragments
    int size = 5;
    int fragment_length = tensor.value.length/size;

    Tensor[] result = new Tensor[size];

    for (int i = 0; i < size-1; i++) {
        result[i] = new Tensor();
        result[i].value =
                Arrays.copyOfRange(tensor.value,
                        i*fragment_length,
                        (i+1)*fragment_length);
    }

    result[size-1].value =
            Arrays.copyOfRange(tensor.value,
            (size-1)*fragment_length,
            tensor.value.length);

    return result;
}
```

```
public static Tensor tensor_multiplication(Tensor tensor, Matrix maxtrix){

    int s = tensor.value.length;
    int x = tensor.value[0].length;
    int y = tensor.value[0][0].length;
    int z = maxtrix.value[0].length;

    Tensor result = new Tensor();
    result.value = new int[s][x][z];

    for (int i = 0; i < s; i++) {
        for (int j = 0; j < x; j++) {
            for (int k = 0; k < z; k++) {
                result.value[s][x][z] = 0;
                for (int l = 0; l < y; l++) {
                    result.value[i][j][k] +=
                            tensor.value[i][j][l] * maxtrix.value[l][k];
                }
            }
        }
    }

    return result;
}
```

```
public static Tensor combine_result(Tensor[] tensors){

    int x = 0;
    for (int i = 0; i < tensors.length; i++) {
        x += tensors[i].value.length;
    }

    int y = tensors[0].value[0].length;
    int z = tensors[0].value[0][0].length;

    Tensor result = new Tensor();
    result.value = new int[x][y][z];

    int index = 0;
    for (int i = 0; i < tensors.length; i++) {
        for (int j = 0; j < tensors[i].value.length; j++) {
            result.value[index] = tensors[i].value[j];
            index++;
        }
    }

    return result;

}


class Tensor{
    public int[][][] value = null;
}

class Matrix{
    public int[][] value = null;
}
```

Code 13 Modified Code

Code 13 is a modified version of Code 12. It has three steps: splitting the original tensor into smaller fragments, calculating tensor multiplications, and combining the calculation result into one tensor. In the new version, the original tensor is split into smaller fragments. The number of fragments can be modified according to the number of spark executors. Thus, the size of spark RDD partitions can be limited to a reasonable number that is not too large and can benefit from the computing cluster's full performance. The internal for-loops are also encapsulated in the additional function. It simplifies the content of the for-loop being translated to avoid unnecessary conflicts and makes the resulting program more stable.

For developers who design using this program, this type of modification is very simple and can make programs more readable. They still do not need to learn the various parallel cloud computing platforms but can deploy their programs on the cloud by using the code translators. It is not difficult but is useful to improve the performance of the resulting program by refactor.

```
//load data
JavaRDD<int[][]> rdd = loadTensor();

int[][] matrix = loadMatrix();

//processing
JavaRDD<int[][]> resultRDD =
    rdd.map(m->{
      int[][] result = new int[m.length][matrix[0].length];

      for (int i = 0; i < m.length; i++)
      {
        for (int j = 0; j < matrix[0].length; j++)
        {
          result[i][j] = 0;

          for (int k = 0; k < m[0].length; k++)
          {
            result[i][j] += m[i][k] * matrix[k][j];
          }
        }
      }

      return result;
    });

//retrieve result
int[][][] resultTensor =  new int[tensor.length][][];

List<int[][]> t = resultRDD.collect();

for (int i = 0; i < tensor.length; i++)
{
  resultTensor[i] = t.get(i);
}
```

Code 14 Manual Translation

Code 14 is the manual translation result. Compared with Code 13, the size of the statements in Code 14 is much smaller than Code 13's because of Spark's functional language programming style. However, it takes a few days to learn this programming language style and Spark development. When performing manual translation, developers do not need to worry about the partition problem because (1) Spark automatically allocates the partition number, and (2) the manual translation code doesn't contain unnecessary variables. There is only one matrix variable shared overall computing nodes.

5.4 Evaluation

Experiments were deployed on a computing cluster on Amazon Web Service (AWS), where each computing node had 8GB memory and 1 process core. Experiments were conducted on five datasets which have a *n*100*1000* tensor and a *1000*1000* matrix and where n is 100, 200, 300, 400, and 500. The test results of four programs (i.e. sequential loop, automatic translation, semi-automatic translation, and manual translation) are described in the following tables and figures.

N	100	200	300	400	500
Sequential(s)	44.818	84.807	123.684	162.955	203.322
Automatic(s)	20.61	35.772	49.912	65.485	79.349
Semi-Automatic(s)	12.345	21.238	31.494	40.083	49.51
Manual(s)	10.156	19.894	28.1	36.667	43.741

Table 8 Time Consumptions of All Translation Results

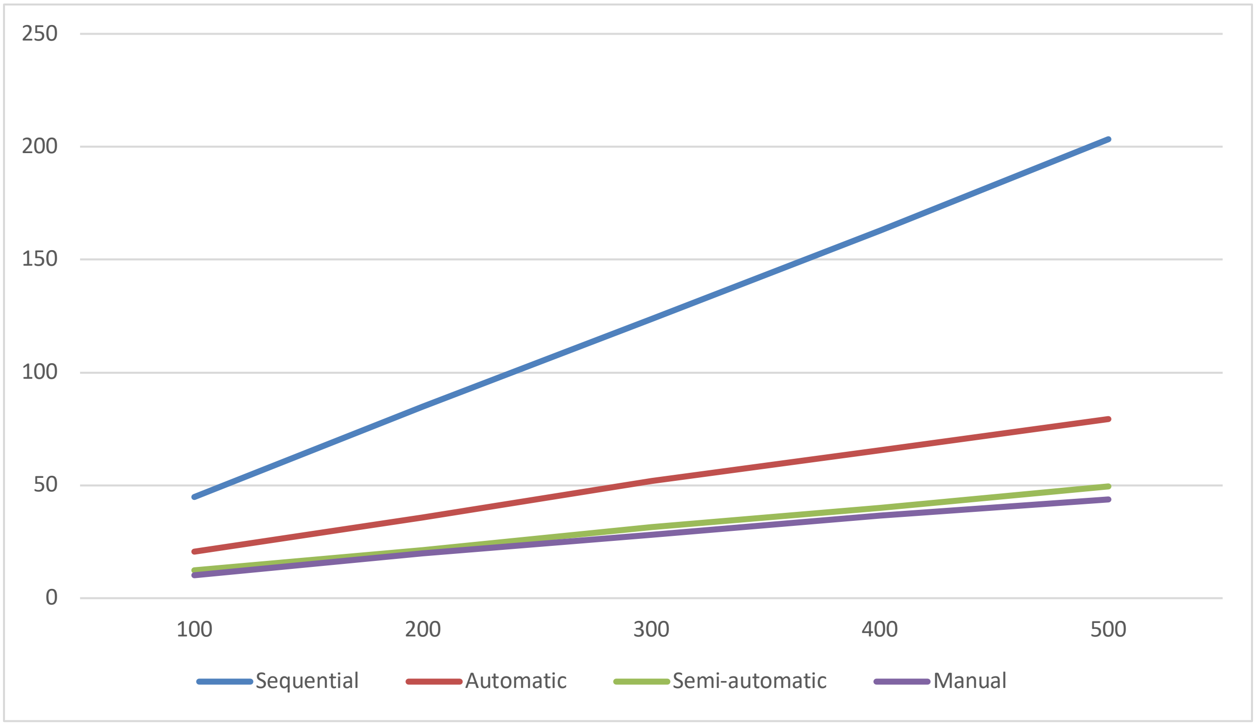

Figure 20 Time consumptions of all translation results

N	100	200	300	400	500
Automatic	2.175	2.371	2.383	2.488	2.562
Semi-automatic	3.63	3.993	3.927	4.065	4.107
Manual	4.413	4.263	4.402	4.444	4.648

Table 9 Speedups of all Translation Results

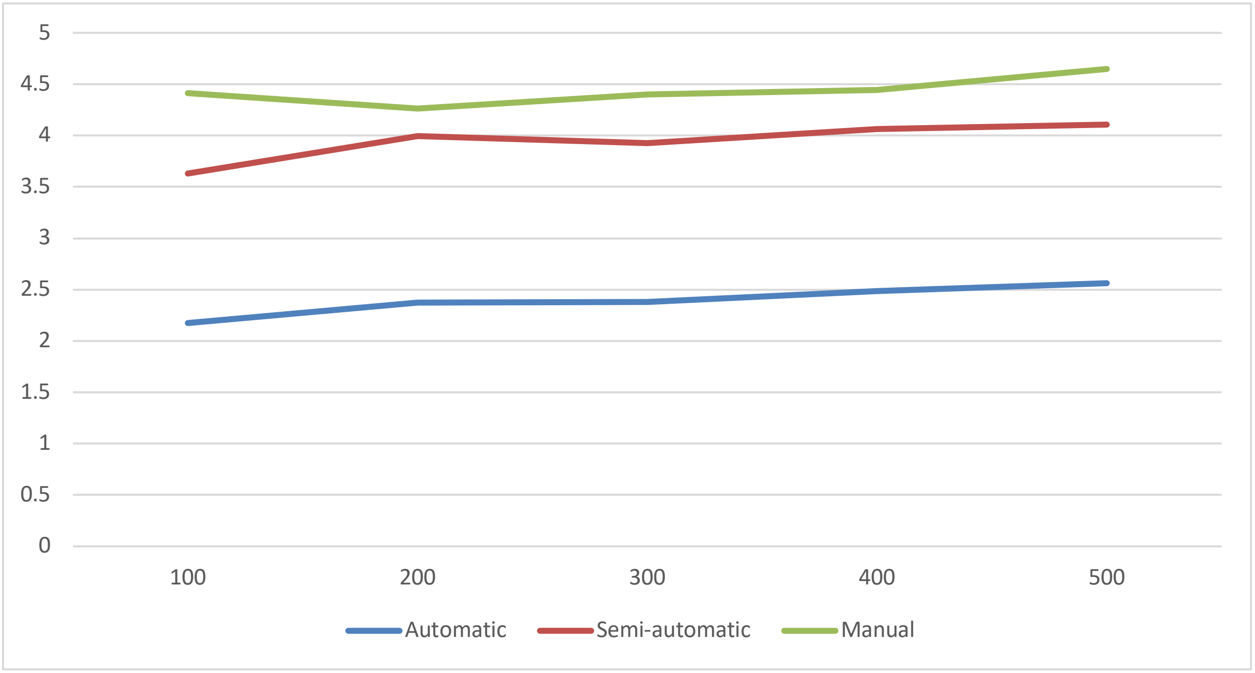

Figure 21 Speedups of all Translation Results

The sequential loop program is the original program without any translation and stated in Code 12. Since the value of n is linear, the time consumption of the sequential program is also linear. In this experiment, all three types of Java to Spark translation are employed, and the speedups of translation results compared to the original sequential program are described in Table 9 and Figure 21.

The first translation result was obtained using automatic translation, meaning it was translated directly by the J2S translator without any modification. The performance is poor since the overhead cost is very large. In the J2S translator, all variables in each iteration of the target loop are encapsulated in an object before the spark process and unloaded from the result object when the process is finished. This design increases the intermediate data size and is significantly costly if the process time of each loop iteration is small enough.

For example, in this test case, the matrix can be shared over all multiplications. It is not necessary to guarantee that each element of a tensor has an individual copy. In Spark applications, an RDD has many partitions, and one partition will not be processed in multiple computing nodes. Specifically, each computing node can process more than one RDD partition, but one partition cannot be split and parallel processed. According to this mechanism, variables can be shared in the RDD partition. Therefore, in the manual translation version, the matrix was only duplicated a few times, which is dependent on the number of partitions. However, in the automatic translation version, it was duplicated N times, where N is the number of matrices in the tensor. It almost doubled memory consumption and the data transmission time.

Compared with automatic translation, semi-automatic translation requires modifying the original loop for iteration size reduction. This modification manually creates data partitions to minimize intermediate-sized data. The original tensor is split into five fragments in this experiment. This operation can significantly reduce the processing time of loading and unloading data. Thus, the resulting program's performance is much faster than the automatic translation result.

The last program is the manual translation result. The performance of the program resulting from manual translation is the benchmark of this application. Compared with automatic and semi-automatic translation, it has two advantages. First, the data can be loaded from the file system or memory directly without encapsulation. Thus, the intermediate data size is smaller than the translated program and can be loaded from the distributed file system. Therefore, loading and unloading data in the manually translated program is significantly faster than programs resulting from the other two translation methods. Second, since no data encapsulation is required, a large number of small-sized partitions can be processed without penalties.

Meanwhile, processing a large number of partitions on the Spark platform is faster than processing a small number of partitions because of the pipeline mechanism. The transmission speed of small data fragments is faster than larger fragments, and the Spark platform can process these fragments without waiting for the whole dataset to finish transmitting. Therefore, the Spark platform can process datasets with small fragments more quickly than those with larger fragments.

Splitting larger data sizes into an appropriate number of partitions can result in better performance. Smaller numbers will increase the overhead and larger numbers will reduce process efficiency. Besides the reasons described above, the number of partitions also impacts the Spark's fault tolerance. If the computing node fails when processing one partition, Spark will retry this partition on the same node or assign a new computing node. Therefore, a larger partition size will increase Spark's recovery cost.

5.5 Conclusion

When all three types of speedups in this scenario are compared, converting a program using semi-automatic translation is faster than using automatic translation but somewhat slower than using manual translation. It shows that automatic translation is not suitable for a large number of small data fragments due to the translation mechanism. However, after some minor modifications of the source code, this type of source code can still benefit from the code translator. This type of modification improves the capability of translators.

This chapter demonstrates a way to improve the performance of a translation program. It is perhaps not the best solution but shows the direction of semi-automatic translation. Since the test tensor in this application only has 100 matrices, it is acceptable to simply split it into n pieces, where n is the number of processing cores in the computing cluster. However, differences between the semi-automatic translation's performance and the benchmarks show that more data fragments can also improve performance in this scenario. Therefore, the preferred method is to balance the number of partitions and the size of each data fragment.

6 CONCLUSION

Over the past several years, I have worked on designing a parallel cloud computing program and existing code migration. I saw that the demand for cloud computing was increasing rapidly. Many researchers or applications must implement cloud computing techniques to handle huge amounts of data. These tasks were mostly impossible to complete previously because of inadequate computing capability.

From these experiences, I was aware that there is a common way to migrate independent iteration statements such as for-loops to parallel cloud applications. Therefore, I designed and developed an automatic Java to MapReduce code translator. It is a prototype of a sequential to parallel computing code translator. There was no translator before the J2M which could directly translate a for-loop statement in a program to a parallel cloud platform application. The J2M translator works but the performance is very limited because the working mechanism of the Hadoop platform is very different from a traditional program.

After some research, I found the Spark platform. Unlike the Hadoop platform, loading files from the file system is not required in the Spark platform. Because of its in-memory mode, it can load and unload data from or to memory. To overcome the J2M translator's constraints, the Java to Spark (J2S) translator was designed and developed. Results of the J2S translator evaluation show that the performance of a program converted using the J2S translator is almost linear in its designed working domain. However, it has its own constraints due to the J2S translator's working mechanism.

Logically speaking, all independent iteration statements can process in parallel. However, due to the working mechanism of the J2S translator, a large number of data partitions can significantly increase the resulting program's overhead cost. Therefore, the semi-automatic translation method was introduced in the last part of my research. It is a way to improve the resulting program's performance by making minor adjustments in the source code. The method is easy but effective. It enhances the capability of the J2S translator in this type of translation.

From the experience of developing these two translators, I found that there is no way to obtain the benchmark performance for automatic translation using the current translator's design. The overhead cost is the tradeoff of automatic translation. I hope we can find better solutions in the future to improve the translators or design some new cloud platforms that can automatically process independent iteration statements in the parallel mode.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "A view of cloud computing," *Communications of the ACM,* vol. 53, no. 4, pp. 50-58, 2010.

[2] J. Bughin, M. Chui and J. Manyika, "Clouds, big data, and smart assets: Ten tech-enabled business trends to watch," *McKinsey Quarterly,* vol. 56, no. 1, pp. 75-86, 2010.

[3] Y. Pan and J. Zhang, "Parallel programming on cloud computing platforms—challenges and solutions," *Journal of Convergence,* vol. 3, no. 4, pp. 23-28, 2012.

[4] N. Yu, B. Li and Y. Pan, "A Cloud-Assisted Application over Apache Spark for Investigating Epigenetic Markers on DNA Genome Sequences," in *2016 IEEE International Conferences on Big Data and Cloud Computing*, Atlanta, 2016.

[5] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha and G. E. Robinson, "Big data: astronomical or genomical?," *PLoS Biol,* vol. 13, no. 7, p. e1002195, 2015.

[6] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak and M. J. Okoniewski, "SparkSeq: fast, scalable, cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics,* p. btu343, 2014.

[7] X. Guo, Y. Meng, N. Yu and Y. Pan, "Cloud computing for detecting high-order genome-wide epistatic interaction via dynamic clustering," *BMC bioinformatics,* vol. 15, no. 1, p. 1, 2014.

[8] R. Kakumani, O. Ahmad and V. Devabhaktuni, "Identification of CpG islands in DNA sequences using statistically optimal null filters," *EURASIP Journal on Bioinformatics and Systems Biology,* vol. 2012, no. 1, pp. 1-14, 2012.

[9] S. Erkek, M. Hisano, C.-Y. Liang, M. Gill, R. Murr, J. Dieker, D. Schübeler, J. van der Vlag, M. B. Stadler and A. H. Peters, "Molecular determinants of nucleosome retention at CpG-rich sequences in mouse spermatozoa," *Nature structural & molecular biology,* vol. 20, no. 7, pp. 868-875, 2013.

[10] H. Wu, B. Caffo, H. A. Jaffee, R. A. Irizarry and A. P. Feinberg, "Redefining CpG islands using hidden Markov models," *Biostatistics,* p. kxq005, 2010.

[11] A. Meissner, T. S. Mikkelsen, H. Gu, M. Wernig, J. Hanna, A. Sivachenko, X. Zhang, B. E. Bernstein, C. Nusbaum and D. B. Jaffe, "Genome-scale DNA methylation maps of pluripotent and differentiated cells," *Nature,* vol. 454, no. 7205, pp. 766-770, 2008.

[12] D. Takai and P. A. Jones, "Comprehensive analysis of CpG islands in human chromosomes 21 and 22," *Proceedings of the national academy of sciences,* vol. 99, no. 6, pp. 3740-3745, 2002.

[13] C. Bock, J. Walter, M. Paulsen and T. Lengauer, "CpG island mapping by epigenome prediction," *PLoS Comput Biol,* vol. 3, no. 6, p. e110, 2007.

[14] M. Hackenberg, C. Previti, P. L. Luque-Escamilla, P. Carpena, M.-A. José and J. L. Oliver, "CpGcluster: a distance-based algorithm for CpG-island detection," *BMC bioinformatics,* vol. 7, no. 1, p. 1, 2006.

[15] D. A. Orlando, M. G. Guenther, G. M. Frampton and R. A. Young, "CpG island structure and trithorax/polycomb chromatin domains in human cells," *Genomics,* vol. 100, no. 5, pp. 320-326, 2012.

[16] N. Yu, X. Guo, A. Zelikovsky and Y. Pan, "GaussianCpG: A Gaussian model for detection of human CpG island," in *2015 IEEE 5th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, 2015.

[17] V. Afreixo, C. A. Bastos, A. J. Pinho, S. P. Garcia and P. J. Ferreira, "Genome analysis with inter-nucleotide distances," *Bioinformatics,* vol. 25, no. 23, pp. 3064-3070, 2009.

[18] N. Yu, X. Guo, F. Gu and Y. Pan, "DNA AS X: An Information-Coding-Based Model to Improve the Sensitivity in Comparative Gene Analysis," in *International Symposium on Bioinformatics Research and Applications*, Springer, 2015, pp. 366-377.

[19] M. Gardiner-Garden and M. Frommer, "CpG islands in vertebrate genomes," *Journal of molecular biology,* pp. 261--282, 1987.

[20] B. Li, J. Zhang, N. Yu and Y. Pan, "J2M: a Java to MapReduce translator for cloud computing," *The Journal of Supercomputing,* vol. 72, no. 5, pp. 1928--1945, 2016.

[21] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[22] T. White, Hadoop: The definitive guide, Sebastopol, CA: O'Reilly Media, Inc., 2012.

[23] J. Talbot, R. M. Yoo and C. Kozyrakis, "Phoenix++: modular MapReduce for shared-memory systems," in *Proceedings of the second international workshop on MapReduce and its applications*, ACM, 2011, pp. 9-16.

[24] B. He, W. Fang, Q. Luo, N. K. Govindaraju and T. Wang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, 2008, pp. 260-269.

[25] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu and G. Fox, "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ACM, 2010, pp. 810-818.

[26] S. Pallickara, J. Ekanayake and G. Fox, "Granules: A lightweight, streaming runtime for cloud computing with support, for map-reduce," in *2009 IEEE International Conference on Cluster Computing and Workshops*, IEEE, 2009, pp. 1-10.

[27] T. Gunarathne, B. Zhang, T.-L. Wu and J. Qiu, "Portable parallel programming on cloud and hpc: Scientific applications of twister4azure," in *Utility*

and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on, IEEE, 2011, pp. 97-104.

[28] J. Lifander and A. Arya, "Automatic conversion of functional sequences to MapReduce with dynamic path selection," 2012. [Online]. Available: https://wiki.engr.illinois.edu/download/attachments/195770312/ABC-2nd.pdf.

[29] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan and U. Srivastava, "Building a high-level dataflow system on top of Map-Reduce: the Pig experience," *Proceedings of the VLDB Endowment,* vol. 2, pp. 1414-1425, 2009.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ACM, 2008, pp. 1099-1110.

[31] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver and J. Zhou, "SCOPE: easy and efficient parallel processing of massive data sets," *Proceedings of the VLDB Endowment,* vol. 1, no. 2, pp. 1265-1276, 2008.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment,* vol. 2, no. 2, pp. 1626-1629, 2009.

[33] R. Lee, T. Luo, Y. Huai, F. Wang, Y. He and X. Zhang, "Ysmart: Yet another sql-to-mapreduce translator," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, IEEE, 2011, pp. 25-26.

[34] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Eltabakh, C.-C. Kanne, F. Ozcan and E. J. Shekita, "Jaql: A scripting language for large scale semistructured data analysis," in *Proceedings of VLDB Conference*, 2011.

[35] L. Lin, V. Lychagina, W. Liu, Y. Kwon, S. Mittal and M. Wong, "Tenzing a sql implementation on the mapreduce framework," Citeseer, 2011.

[36] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz and A. Rasin, "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads," *Proceedings of the VLDB Endowment,* vol. 2, no. 1, pp. 922-933, 2009.

[37] K. Bajda-Pawlikowski, D. J. Abadi, A. Silberschatz and E. Paulson, "Efficient processing of data warehousing queries in a split execution environment," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ACM, 2011, pp. 1165-1176.

[38] J. Zhang, D. Xiang, T. Li and Y. Pan, "M2M: A simple Matlab-to-MapReduce translator for cloud computing," *Tsinghua Science and Technology,* vol. 18, no. 1, pp. 1-9, 2013.

[39] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE computational science and engineering,* vol. 5, no. 1, pp. 46-55, 1998.

[40] D. Borthakur, "HDFS architecture guide," *Hadoop Apache Project,* vol. 53, 20008.

[41] B. Li, X. Xiao and Y. Pan, "Automatic translation from Java to Spark," *Concurrency and Computation: Practice and Experience,* 2018.

[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker and I. Stoica, *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation,* pp. 2--2, 2012.

[43] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation,* pp. 2--2, 2012.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud,* vol. 10, no. 10, p. 95, 2010.

[45] "berkeley.edu," [Online]. Available: http://hitoshi.berkeley.edu/221a/tensorproduct.pdf. [Accessed 28 1 2018].

[46] P. J. Basser, J. Mattiello and D. LeBihan, "MR diffusion tensor spectroscopy and imaging," *Biophysical journal,* vol. 66, no. 1, pp. 259--267, 1994.

[47] D. Le Bihan, J. Mangin, C. Poupon, C. A. Clark, S. Pappata, N. Molko and H. Chabriat, "Diffusion tensor imaging: concepts and applications," *Journal of magnetic resonance imaging,* vol. 13, no. 4, pp. 534--546, 2001.

[48] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review,* vol. 51, no. 3, pp. 455--500, 2009.