J Supercomput
DOI 10.1007/s11227-016-1695-x

J2M: a Java to MapReduce translator for cloud computing

Bing Li[1] · **Junbo Zhang**[2] · **Ning Yu**[1] · **Yi Pan**[1]

Abstract Cloud computing has gradually evolved into an infrastructural tool for a variety of scientific research and computing. It has become a trend that lots of products have been migrated from local servers to cloud by many institutions and organizations. One of the challenges in cloud computing now is how to run software efficiently on cloud platforms since lots of original codes are not capable of being executed in parallel on cloud contexts, resulting in that the power of clouds cannot be exerted well. It is costly to redesign and convert current sequential codes into cloud platform. Thus, automatic translation from sequential code to cloud code is one of the directions that could be taken to resolve the problem of code migration in cloud infrastructure. In this paper, a new *Java to MapReduce* (J2M) translator is developed to achieve the automatic translation from sequential Java to cloud for specific data-parallel code with large loops. This paper will provide details about the design of our translator and evaluate our performance through experiments. The experimental results not only indicate that the translator can precisely translate the sequential Java into cloud codes, but also show that it can achieve very good speedup in performance, and we expect that an almost linear speedup is possible if larger enough data is processed. It is believed that the J2M translator is an ideal stereotype for code migration and will play an important role in the transition era of cloud computing.

Keywords Java · MapReduce · Cloud computing · Code migration

✉ Yi Pan
pan@cs.gsu.edu

1 Department of Computer Science, Georgia State University, Atlanta, Georgia 30303, USA

2 School of Information Science and Technology, Southwest Jiaotong University, Chengdu 610031, China

1 Introduction

Cloud computing is a new architectural model that provides sufficient computing resources (hardware and software) to users in an economical way. It has been proven very successful in big data applications such as database applications and searching engines. The fundamental step is to move jobs from conventional servers to shared cloud servers. Cloud servers provide superior services to users instead of the service provided by personal computers or even small servers [2]. The two major advantages of cloud computing are scalability and efficiency. They bring more powerful and more economical computing models to users because of its large scale and resource sharing. Users only need to pay for the time and the assigned computing capability rather than purchasing expensive hardware equipment. More institutions and organizations have moved their work to cloud servers. It leads to increasing demands of skilled software engineers on cloud computing [5]. Also, expanding the horizon of cloud computing systems for more applications executed on cloud is another 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. How to effectively implement parallel codes on cloud is still an issue [18]. Programs developed for conventional platforms lack the parallel capability on cloud even when the hardware platforms are updated. Three feasible solutions for the transition of cloud computing are mirror/copying, de novo development, and automatic translation. First of all, mirror/copying is a simple method to deploy local applications to a cloud system. Obviously its cost of migration is low but its performance can barely be promoted due to the lack of taking advantage of the parallel architecture. Second, according to Cloud programming model, de novo development is expensive in cost. Among many emerging programming models for cloud platforms, MapReduce has become one of the most popular programming models for big data analysis in cloud systems due to its simplicity in implementing data parallel applications [9], such as Apache Hadoop [21], Phoenix [19], Mars [13], Twister [10], and Granules [17].

- Apache Hadoop (Hadoop for short in the following content) [21] was developed for data-intensive distributed applications.
- Phoenix [19], 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 [13], a GPU implementation of MapReduce, provides a generic framework for developers to implement data- and computation-intensive tasks correctly, efficiently, and easily on the graphic processors (GPUs). It hides the programming complexity of the GPU behind the simple and familiar MapReduce interface. Hence, the developers can program on the GPU without any knowledge of the GPU architecture or the graphics APIs.
- Twister [10], an implementation of Iterative MapReduce model, is a lightweight runtime. It provides the feature for cacheable MapReduce tasks and allows programmers to develop iterative applications without spending overmuch time on reading and writing large amounts of data in each iteration. Twister4Azure [12], a version for Windows Azure, has also been released.

- Granules [17] 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 is increasing, Hadoop grows ever more popular. Hadoop provides a great platform for logistics and coordination such as task failure, loading input, splitting input, and task coordination. It makes programming job easy. Each job we come across in MapReduce presents an applicable way to resolve the problem with the amount of data increasing tremendously. However, it is not easy to learn how to program in MapReduce because the MapReduce paradigm is in low level and rigid. To develop the applications with these runtime systems, the developers should be proficient in the following three domains: MapReduce model, runtime system, and system-specific programming language. Although many applications can be parallelized by this method, the developers still need to be trained hard in the above three domains. Thus, the transition to cloud programming for developers is torturous. It will be desirable if there is an alternative way to utilize the Hadoop platform without learning MapReduce programming.

Third, the automatic translation from the targeted code to the desired cloud computing model is a feasible way for code migration. Considering the huge demand of cloud computing, many institutions move their products from local equipment to a cloud server. In order to redesign and transform the current products into cloud computing model, whether to hire new engineers or retrain current employees, the employers can spend a lot. Apparently, if a new way is invented to convert existing local programs to cloud programs automatically, it would be well received in the cloud computing community.

Our research aims to explore the automatic conversion from sequential codes into cloud codes by targeting at Java program, a most popular programming languages in use.[1] Moreover, the most popular cloud computing platform Hadoop is written by Java. These facts prompt us to develop a Java to MapReduce (J2M) translator, a new way for solving the two problems mentioned in mirror/copying and de novo sections. Due to the complexity of the problem, we will first focus on codes with simple large loops without data dependency. Later on, we will investigate more complex loop structures.

In the following, Sect. 2 shows relevant work connected with our research. Section 3 describes the basic structure of the proposed translator and the translation procedure. Section 4 presents two experiments to evaluate the performances of J2M and assesses the empirical results. The last section is about the conclusion and the future work.

2 Related work

The manual conversion from sequential code to cloud code is a time-consuming, error-prone procedure to redevelop a cloud application, because the data flowing in and out of the MapReduce code must be carefully and precisely coordinated with the other sequential sections of the code [15]. For example, a Hadoop programmer has to 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

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

is a certain 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 [11,16], SCOPE [6], Hive/HiveQL [20], YSmart [14], YSmart-S, Jqal [4], Tenzing [7], HadoopDB [1,3], and SQL-to-MangoDB, etc.

In what follows, we will review some representative works.

2.1 Hive/HiveQL

Hive [20] 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 SQL-like language, which is called HiveQL. It enables the users who are familiar with SQL to learn HiveQL in a relatively short time and perform 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. In Facebook, there are more than 700TB of data and tens of thousands of tables stored in the Hive warehouse, which is utilized by over 200 users each month.

2.2 HadoopDB

HadoopDB [1] is a hybrid system of MapReduce and DBMS technologies. There are four main design features of HadoopDB: designed to run on a shared-nothing cluster of commodity machines or in the cloud, designed as an attempt to fill the gap in the market for a free and open source parallel DBMS, designed for greater scalability than currently available parallel database systems and DBMS/MapReduce hybrid systems, and designed to be as scalable as Hadoop, while achieving superior performance on structured data analysis workloads.

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

- *Database connector* This is the interface between TaskTrackers 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 databases. 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 systems.
- *Query interface* This allows queries to be submitted via a SQL/MapReduce API.

The SQL to MapReduce to SQL (SMS) planner has extended Hive from two areas: (I) updating the MetaStore with references to our database tables before any query execution; (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 according 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 data and provide automatic fault tolerance and load balancing [3].

2.3 YSmart: SQL-to-MapReduce translator

YSmart [14] 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 computation, I/O operation and network overhead. There are some results demonstrating that YSmart outperforms Hive and Pig on correlated queries execution.

The contribution YSmart claimed is building a correlation-aware SQL-to-MapReduce translator for improving the complex query performance without modifying the underneath Hapdoop MapReduce system. There are limitations of MapReduce for complex queries. For instance, local disk's copies of intermediate results are needed for node failure and temporary results are required to 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 complex 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. The significant optimization is implemented via YSmart job merging. YSmart recongnizes the intra-query correlation and provides a set of rules for multiple job merging. These rules are well defined to handle different cases. The details can be found in [14]. 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 of above three projects are SQL-to-MapReduce applications and limited to SQL queries translation. The following two projects are the implementation of computation statement translation.

2.4 M2M translator

These aforementioned languages and translators inspired researchers Zhang et al. to develop the translator M2M [22] in the year 2013, which can translate Matlab codes to MapReduce codes. Matlab is a popular programming language in the scientific research area 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 background knowledge. Moreover, users do not need spend much time on programming. For instance, users can call functions from standard Matlab library directly, when they need to operate matrices like cross product and dot product. In other words, the main part of Matlab program is a combination of standard functions. Therefore, the major calculation jobs of Matlab program are processed in the standard library functions.

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

2.5 OpenMP

OpenMP is famous industry level APIs for C/C++ and Fortran programming [8]. It has already become a standard of most C/C++ and Fortran compilers. When users implement OpenMP in the source codes, the compiler can convert target parts of codes highlighted by users to multi-thread models. Unlike M2M translator, which is unavailable to translate users functions, OpenMP can process any type of independent codes. Regarding dependency problems, it provides methods to convert partial dependent statements to the independent models. OpenMP was developed in the year 1997 when the cloud computing was not invented, so that it fails to apply to the cloud computing program translation. Therefore, a new sequential program to cloud program translator (J2M) is proposed in this paper. It implements partial features of OpenMP, and works on the cloud program translation as well.

3 J2M translator

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

The structure of the proposed translator is shown in Fig. 1. The translator has four major components: scanner, parser, extractor, and generator. They are responsible for four translation procedures, respectively, namely tokenization, parsing, extraction, and generation. It also has a minor component termed Name Database, which can help Parser and Extractor to record the variables' name.

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. Subsequently, the extractor extracts all necessary information from the objects list, packs them, and pass this package to the generator. At the end, 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 Fig. 2.

Unlike a typical compiler, which has only two stages: tokenization and parsing, the J2M translator translates Java source files through four stages: tokenization, parsing,

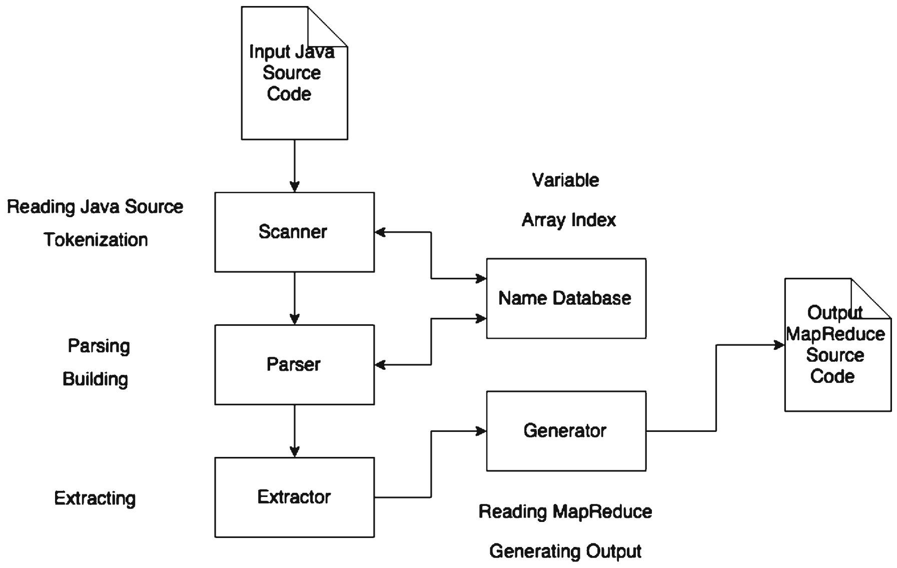

Fig. 1 J2M translator structure

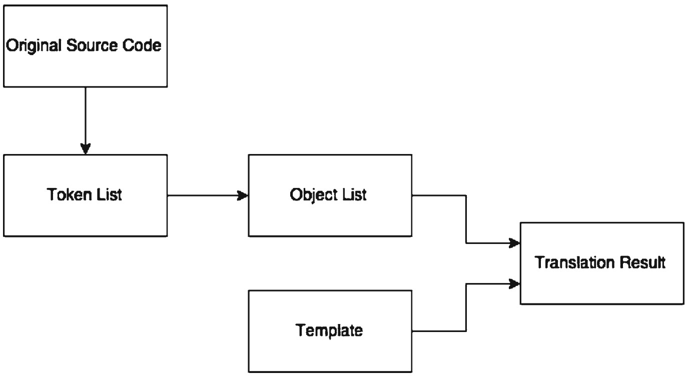

Fig. 2 Translation procedures

extraction, and generation. This section provides details of each stage in the translation procedures, followed by a translation example used for the illustration of each stage.

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 tokens values by string objects, but also records the types of these tokens, which is an additional job of tokenization.

Table 1 Token list

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

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

The *Value* token is a set of string value 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 *Identifier* and converted to *Value* in the parsing stage. It is not necessary to separate these values into different groups because of two reasons. The first reason is that all input Java source files should be checked by Java compiler above all without compile errors. Therefore, the translator skips the syntax check for this input source file. The other reason is that 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 semicolon, comma, and period. Similar to *Value* tokens, the proposed translator does not parse these symbols, which may help the translator to analyze identifiers in parsing stage.

The *Directive* is a mark to highlight the target loop for J2M translator. There are only two types of directives, *//J2MSTART* and *//J2MEND* for start and end, respectively. Since directives use 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 *EOF* token appears at the end of token list. It is not extracted from input Java source file but generated by the scanner automatically. It highlights that the token list is terminated.

In a real instance, 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

```
Token List:

...
//J2MSTART : Directive
for        : Identifier
(          : Operator
int        : Identifier
i          : Identifier
=          : Operator
0          : Value
...
EOF        : EOF
```

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 variable but also records the type information. Unlike the parsing stage in regular compiler, analyzing entire codes or entire instructions is unnecessary. The proposed translator analyzes only the single token.

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 tokenization stage. The parser simply converts them from string tokens into objects without any additional features.

Table 2 Object list

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

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 words 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: byte, int, short, long, boolean, float, double and char. The additional one is defined as object type. The *Parser* analyzes all object types in declaration instructions, variable declaration and method declaration, and then converts them into *Type* objects.

The *Variable* object is the successor of *Identifier* token. It is declared in the declaration instruction and then used in anywhere afterwards. The parser needs to analyze the type of all variables, which contains two cases. One is inside the variable declaration and the other one 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 totally 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 when it meets a variable outside of declaration, it searches in the variable stack to retrieve the type of that variable.

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

ClassName and *MethodName* objects work together for they are both special variables. In our translation result, the method name always contains its class name. To achieve this feature, each time when the parser detects a method declaration, it searches 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 *Imports* object are also very important parts of codes. They contain the package name of 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 token list provided in 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 Extraction

To generate the translation result, the J2M translator implements a MapReduce function template. More details about this area 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 input source code. The objective of the extractor is to prepare all necessary information for generating 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.

Actually, there are three parts of variable operations in the template. The first part is declaration. The extractor scans target loop and collects all variable information. After that it creates the declaration list and converts array variables into regular variables simultaneously. It is unnecessary to declare array variables in the translation result.

The last two parts are data import and export. The translation result should be executed on Hadoop cluster, which regulates that all variables are loaded from 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: the first one is to convert array variables into regular variables declared in the variable declaration part; the second one is to move everything from target loop to the result function.

Following the example object list in the last stage, the information can be extracted as what is 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 codes, *a[i]* and *a[i+1]*, were transformed to regular variables, *array_a_0* and *array_a_1*, in the stage of parsing.

3.4 Generation

The generation stage is simpler than previous three stages. The *Generator* combines the result of extraction stage and the template to generate a final translation result code. The most important part is the design of MapReduce function template. The following is our temple 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,
            included data import
            and export*/
        }
    }
}
```

Compared to the OpenMP, which can convert some dependent loops to independent loops, currently, the J2M translator only supports independent loops. Although the first generation of J2M translator is still in its trial period, more practicable features will be added to in order to make it sophisticated in the future. The constraints of J2M translator will be introduced in result 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 original loop using a new *while* loop. Each Mapper function can execute multiple steps of original loop. The total number of Mapper function will depend on the environment variable. If the total step of original loop is M and the total number of Mapper is N, the steps in each Mapper function should be $\frac{M}{N}$.

In this template, there are four highlight areas: package name, import classes, variable declaration and body. In the generation stage, these four areas can be filled by the information that retrieved from the extractor.

The final result of the example code is as follows:

```
package ...
import ...
... class and method headers ...

int i;
int array_a_0;
int num;
int array_a_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));
...
```

4 Evaluation and assessment

After user translates a Java file by using the J2M translator, two facts are concerned by users: computing correctness and performance speedup. One of the most important features of the translation result is performance, especially the execution speed. If the execution speed of translation result is slower than original Java program, the proposed translator is useless. In order to test the performance of the J2M translator, experiments on those two concerns were created. The experimental results not only indicate that the translator can precisely translate the sequential Java into cloud codes, but also show that it can achieve very good speedup in performance, and we expect that an almost linear speedup is possible if larger enough data is processed. Due to memory limitation, larger data cannot be processed on our current facility. This section provides the details of these experiments.

4.1 Constraints

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

- Currently, the J2M translator can only work on Java *for* loop and translate only one loop each time.
- The target loop should be highlighted by J2M directives by users.
- The target loop should be located in a static method instead of object method.
- No nested loop is available in current version of translator.
- The target loop should be an independent loop.
- Users can not declare any array inside the target loop. However, they can declare them in other place and use them inside the loop.

4.2 Sequential java program vs MapReduce program

It is for two reasons that we use cloud servers instead of local machines. The first one is that the limitation of computing resource. For example, local computers may not have sufficient memory or hard drive capacity to handle big data problems. Since users cannot load terabytes data into computer memory, they have to use cloud computing technique, even though local computer might be faster than any node of server. Normally, if the size of data is not huge enough, the programs executed on local computers can be faster than on cloud servers because communication and preheating costs of cloud computing technique may be expensive. Therefore, the experiments are designed to evaluate of non data-intensive cases.

The other reason is that the cloud server can be faster than local computer when executing compute-intensive programs. For example, if a user executes a program on a N nodes cloud server, it takes N times faster than local computers in the extreme case. However, it cannot be processed in reality due to the intercommunication cost.

This experiment measures the performance of the translation result in a huge number of floating point calculations, 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 Fig. 3, where the cloud program is faster than local program.

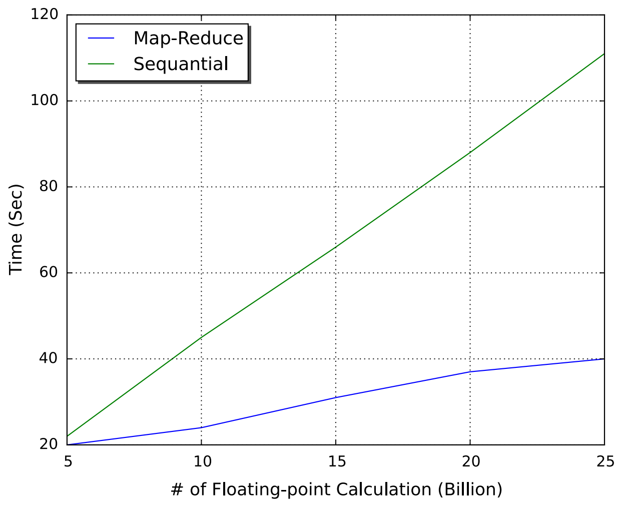

Fig. 3 Performance result

Table 3 Test 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 represents the time costs of five test cases. In cloud server, the warming-up cost is the reason why the time cost increment is less than 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 in different servers. Therefore, the analysis should focus on the increment of time cost only.

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

4.3 Performance on different calculation size

Comparing sequential Java program and MapReduce program, the major disadvantage of MapReduce algorithm, especially on cloud server, is that the MapReduce program spends much time on communication. It is why the sequential programs are faster than parallel programs on small-size calculation. In the previous experiment, the MapRe-

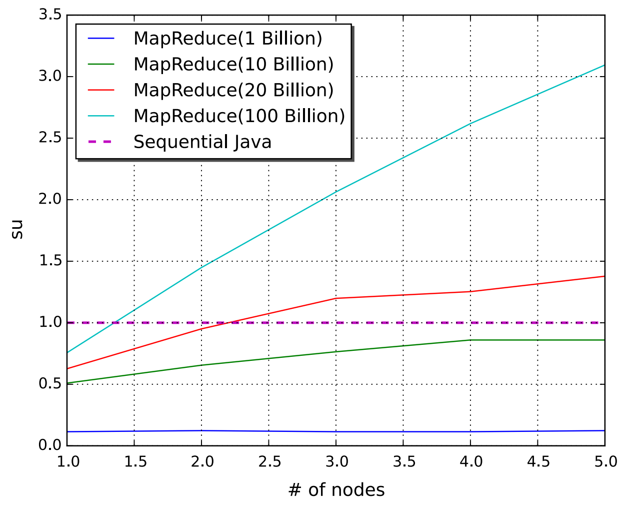

Fig. 4 Performance on different calculation size

duce program was faster than local program in all test cases because the calculation size is big enough. However, what is the performance of the translation results on different calculation size, especially on small-size computing? 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 that how the MapReduce program is faster than sequential Java program. The formula of SU shows following:

$$SU = \frac{\text{Time Cost of Sequential Java Program}}{\text{Time Cost of MapReduce Program}} \tag{1}$$

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

In this experiment, the test cases use four different calculation sizes: one billion, ten billion, twenty billion, and one hundred billion. The MapReduce program uses one to five nodes on cloud server in each one of five cases. The results are shown in Fig. 4. 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 that in these two test cases the MapReduce programs are slower than sequential Java programs. The yellow line is above all other lines. The red line is somewhere between yellow line and green line.

Compared to the blue line, the green line speeds up once the number of nodes is increasing. On the contrary, there is no obvious change on blue line. The reason is that when the calculation size is small enough, the time cost of communication is equal to or greater than the speed up benefits in which the number of cloud server nodes increasing. The other evidence is that the increments of red and green lines are decreasing and almost become to 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 sever node. Besides the communication cost, there is a warming-up time cost. Cloud computing platforms like Hadoop need a certain time to start up and to initialize the environment and load data. It is an alternative factor that affects the execution time of cloud programs.

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

5 Conclusion

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

References

1. Abouzeid A, Bajda-Pawlikowski K, Abadi D, Silberschatz A, Rasin A (2009) HadoopDB : an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc VLDB Endow 2(1):922–933. http://dl.acm.org/citation.cfm?id=1687627.1687731
2. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I et al (2010) A view of cloud computing. Commun ACM 53(4):50–58
3. Bajda-Pawlikowski K, Abadi DJ, Silberschatz A, Paulson E (2011) Efficient processing of data warehousing queries in a split execution environment. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, SIGMOD '11, pp 1165–1176. ACM, New York, NY, USA. doi:10.1145/1989323.1989447
4. Beyer KS, Ercegovac V, Gemulla R, Balmin A, Eltabakh MY, Kanne CC, Ozcan F, Shekita EJ (2011) Jaql: a scripting language for large scale semistructured data analysis. PVLDB, pp 1272–1283
5. Bughin J, Chui M, Manyika J (2010) Clouds, big data, and smart assets: ten tech-enabled business trends to watch. McKinsey Q 56(1):75–86
6. Chaiken R, Jenkins B, Larson PA, Ramsey B, Shakib D, Weaver S, Zhou J (2008) SCOPE : easy and efficient parallel processing of massive data sets. Proc VLDB Endow. 1(2):1265–1276. http://dl.acm.org/citation.cfm?id=1454159.1454166

7. Chattopadhyay B, Lin, L, Liu W, Mittal S, Aragonda P, Lychagina V, Kwon Y, Wong M (2011) Tenzing a SQL implementation on the MapReduce framework. In: Proceedings of VLDB, p 1318–1327
8. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. Comput Sci Eng IEEE 5(1):46–55
9. Dean J, Ghemawat S (2008) MapReduce: simplified data processing on large clusters. Commun ACM 51(1):107–113. doi:10.1145/1327452.1327492
10. Ekanayake J, Li H, Zhang B, Gunarathne T, Bae SH, Qiu J, Fox G (2010) Twister : a runtime for iterative MapReduce. In: Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, HPDC '10, ACM, New York, NY, USA, p 810–818 doi:10.1145/1851476.1851593
11. Gates AF, Natkovich O, Chopra S, Kamath P, Narayanamurthy SM, Olston C, Reed B, Srinivasan S, Srivastava U (2009) Building a high-level dataflow system on top of Map-Reduce: the Pig experience. Proc VLDB Endow. 2(2):1414–1425. http://dl.acm.org/citation.cfm?id=1687553.1687568
12. Gunarathne T, Zhang B, Wu TL, Qiu J (2011) Portable parallel programming on cloud and HPC: Scientific applications of twister4azure. In: UCC'11, p 97–104
13. He B, Fang W, Luo Q, Govindaraju NK, Wang T (2008) Mars : a mapreduce framework on graphics processors. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, PACT '08, ACM, New York, NY, USA, pp 260–269. doi:10.1145/1454115.1454152
14. Lee R, Luo T, Huai Y, Wang F, He Y, Zhang X (2011) YSmart: Yet another SQL-to-MapReduce translator. In: Distributed Computing Systems (ICDCS), 2011 31st International Conference on, p 25 –36. doi:10.1109/ICDCS.2011.26
15. Lifander J, Arya A (2012) Automatic conversion of functional sequences to MapReduce with dynamic path selection. https://wiki.engr.illinois.edu/download/attachments/195770312/ABC-2nd.pdf
16. Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of data, SIGMOD '08, ACM, New York, NY, USA, pp 1099–1110. doi:10.1145/1376616.1376726
17. Pallickara S, Ekanayake J, Fox G (2009) Granules: A lightweight, streaming runtime for cloud computing with support, for Map-Reduce. In: Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on, pp 1–10. doi:10.1109/CLUSTR.2009.5289160
18. Pan Y, Zhang J (2012) Parallel programming on cloud computing platforms. J Converg 3(4):23–28
19. Talbot J, Yoo RM, Kozyrakis C (2011) Phoenix++ : modular MapReduce for shared-memory systems. In: Proceedings of the second international workshop on MapReduce and its applications, MapReduce '11, ACM, New York, NY, USA, p 9–16. doi:10.1145/1996092.1996095
20. Thusoo A, Sarma JS, Jain N, Shao Z, Chakka P, Anthony S, Liu H, Wyckoff P, Murthy R (2009) Hive: a warehousing solution over a Map-Reduce framework. Proc VLDB Endow. 2(2):1626–1629. http://dl.acm.org/citation.cfm?id=1687553.1687609
21. White T (2010) Hadoop: The Definitive Guide, 2nd edn. O'Reilly Media, Inc., Sebastopol, CA
22. Zhang J, Xiang D, Li T, Pan Y (2013) M2M : a simple Matlab-to-MapReduce translator for cloud computing. Tsinghua Sci Technol 18(1):1–9