Received: 14 September 2017 | Revised: 4 December 2017 | Accepted: 26 January 2018
DOI: 10.1002/cpe.4459

WILEY

SPECIAL ISSUE PAPER

Automatic translation from Java to Spark

Bing Li | **Xueli Xiao** | **Yi Pan**

Department of Computer Science, Georgia State University, 25 Park Pl NE, Atlanta, GA 30303 USA

Correspondence
Yi Pan, Department of Computer Science, Georgia State University, 25 Park Pl NE, Atlanta, GA 30303 USA.
Email: yipan@gsu.com

Present Address
Yi Pan, Department of Computer Science, Georgia State University, 25 Park Pl NE, Atlanta, GA 30303 USA.

Summary

Cloud computing is a powerful infrastructure tool for scientific computing and research and attracts many developers to deploy their products on it. Code migration from local to cloud servers is a significant challenge for all companies and organizations that have many legacy programs. Compared to the de novo development method, the automatic translation is more productive and economical. In this paper, a new Java to Spark (J2S) translator is introduced to achieve the automatic translation from sequential Java code to Spark cloud application, which can translate three types of computing intensive programs. The evaluation experiments demonstrate that these translations significantly improve the source program performance in their domains. It is believed that it is a new step in the automatic code migration era of cloud computing.

KEYWORDS
cloud computing, code migration, Java, MapReduce, Spark

1 | INTRODUCTION

Cloud computing is a modern computing resource management system, which centralizes both data storage and computing resource. It refers to both the hardware layer and the applications on this hardware system.[1] Cloud server is the hardware system in cloud computing, which contains process and storage utilities. It can be either a large public data center for any clients, such as Amazon AWS, Google GCP, and Microsoft Azure, or a small private server.[2] Clients benefit from cloud servers' powerful and secure service. A cloud application processes its primary computing tasks and stores data on cloud servers to reduce the work load on the client side.[3,4]

The demand for cloud computing increases rapidly in recent years because an increasing number of companies move their products from local servers to cloud servers to reduce the maintenance cost of their products.[5] In this procedure, a new question is raised, which is how to reduce the migration cost of their legacy codes. Unlike new companies that can develop their products on cloud directly, a company with a long history can have a huge number of legacy codes that use old programming models. It is a big problem because moving these products to the cloud will cost too much for the companies to afford.

To maximize the performance of cloud applications, the code migration method needs to not only implement the original programs on the cloud server but also transform their programming model from sequential to parallel.[6] This transformation is the most difficult part of the code migration procedure because it requires analyzing data dependency and then refactoring the source code. Sometimes, it is impossible to transform the legacy code due to data dependency and the costs.

The automatic translation method is a possible solution to the sequential-to-parallel code translation problem, which can significantly reduce the labor cost. Automatic migration tools are often built in compilers. When a programming language has significant syntax updates, they help the compilers to compile the previous version code without any modification. However, translating code from one programming model to another is more complicated. For example, OpenMP is a famous code translator, which translates C/C++ and Fortran code from sequential to parallel. In the translation procedure, developers must handle the data dependency and insert directives for OpenMP by themselves. In other words, OpenMP is not responsible for data dependency detection.

Currently, the major local-to-cloud code migration method is to develop a new platform with a similar application interface (API), such as Hive,[7] a SQL database on Hadoop[8] Platform. Developers can easily migrate their SQL queries between different SQL databases due to the similar interface. Besides these platforms, some translators can also help developers to migrate their code to these platforms such as YSmart,[9] a SQL to MapReduce[10] translator, and M2M,[11] a MATLAB to MapReduce translator. Both translators translate source code by replacing the standard API functions with their substitutes that work on cloud platforms. In other words, they can only translate the functions that are in their substitute function list.

J2M[12] is a Java to MapReduce translator inspired by OpenMP and M2M projects, which can translate a simple Java for-loop that does not contain data dependency to MapReduce program. Similar to OpenMP, the J2M translator also requires developers to insert directives to mark the loops that will be translated. Unlike the M2M translator, the J2M translator does not prepare substitute functions but provides a MapReduce template for the loop translation.

Since the output program of the J2M translator is a MapReduce application running on Hadoop Platform, the intermediate data of the source program must upload to and download from the 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. And then, 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. In the end, users need to manually download the result file from the HDFS and execute the program after the loop that will read result data from the downloaded file and continue the remaining tasks.

This method is inconvenient because it requires developer participation. At the same time, the HDFS operations 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. If the intermediate data is large enough, the performance of the translation result may be worse than the source one. To avoid this problem, executing the resulting programs on an in-memory cloud computing platform is a possible solution.

Java to Spark (J2S) translator is proposed in this paper, which is built on the J2M translator's framework. Compared with J2M translator, the J2S translator generates Spark program instead of 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,[13] the J2S translator can communicate with Spark platform directly without generating the 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 during the program running time.

On the other hand, the in-memory model takes more memory space than the HDFS one. Therefore, the capability of the J2S translation depends on the memory capacity of the working environment, the computing cluster where the resulting program is deployed. In other words, this system does not work if the intermediate data size is larger than the memory capacity. Meanwhile, the in-memory model also increases the hardware cost because the large memory system costs more. All of these are the trade-off of implementing the in-memory model. Developers should analyze their use cases before using the J2S translator.

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

In the following, Sections 2, 3, and 4 describe the design and evaluation of for-loop parallelization, task parallelization, and for-loop-in-task parallelization, respectively. The last section is the future work and conclusion of the proposed project.

2 | FOR-LOOP PARALLELIZATION

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

Figure 1 describes the program structure of a translation resulting program, containing two major parts, Main Program and Data Container. All three data process stages are in Main Program. In the Data Preparation stage, all intermediate data are packaged and submitted to Spark Resilient Distributed Datasets (RDD). 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 the communication between local memory and Spark RDD.

2.1 | 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 entity of RDD can hold only one data object. However, the intermediate data may have a significant number of variables. Thus, these variables need to be packaged into one object.

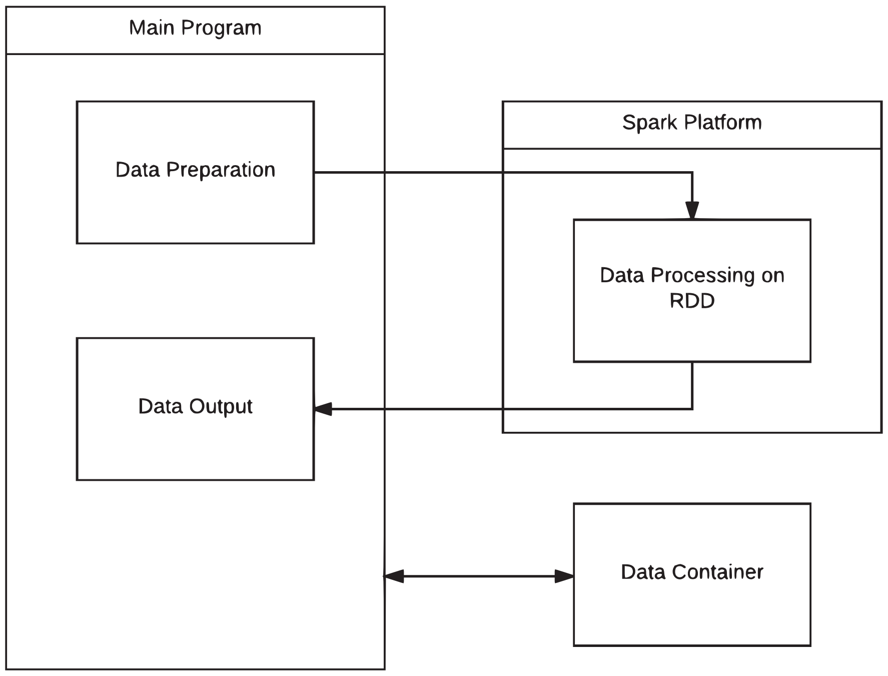

FIGURE 1 The structure of the for-loop translation resulting program

```
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]);
    }
}
```

Listing 1 A source program example

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

}
```

Listing 2 A data container example

Listing 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 Listing 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 *LoopData* class for handling all variables in the targeted for-loop. The *num[i]* variable is one element in the *num* array. Since there is only one element that is

```
//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();
```

Listing 3 A data preparation program example

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 for replacing the *num[i]* variable in the Data Container class.

2.2 | Data preparation

Data Preparation is the first step of processing data on Spark platform. The code piece in Listing 3 is the Data Preparation stage translated from the example Java program in Listing 1. RDD initialization and Spark configuration are at this stage.

A Data Container list is declared in 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 when it only controls the number of iterations but does not participate in data processing. However, to simplify the translation, the proposed translator is designed to neglect detecting whether the iterator is useless or not, 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 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 instead of configured separately.

2.3 | Data processing

The Data Processing stage is the main procedure in the translation resulting program. Listing 4 is an example of the data process stage in the translation result. The Spark Map function in this stage is implemented by migrating of instructions in the original for-loop. It is then processed on the Spark cluster. In other words, the original for-loop is implemented as distributed processes in this stage.

There are three steps in the Map function, which are data loading, processing, and storing. In the first step, ie, 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 for holding the corresponding values. Second, these data are processed in a statement that is migrated from the original for-loop. This migration is no more than simple copy-and-paste, leaving array variables as the only thing needed to be modified, such as the *num_0* variable in Listing 4. Since only partial elements of an array are required to be processed in an iteration, they are converted to non-array variables.

In the end, the data in the Spark RDD are required to be updated. This step simply assigns values of all temporary variables to the data container, which is opposite to the loading step in the Spark Map function. To simplify this procedure, all variables are updated without modification check. In other words, whether the variable is modified or not, the update procedure is always carried out.

2.4 | Data output

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

```
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;
}});
```

Listing 4 A data process program example

```
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;
}
```

Listing 5 An output program example

2.5 | Evaluation

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

Figure 2 shows that the time cost of the parallel program decreases as the number of cores grows. The last row in Table 1 demonstrates the speed-up of the Spark program, whose values are calculated by dividing the time cost of the sequential program over that of the parallel one. Theoretically, the optimal speed-up should be equal to the number of computing cores used in the experiment. However, in actual practices, it is impossible to reach the maximum value due to the overhead costs. The speed-up values in Table 1 are quite reasonable because they are close to the optimal ones.

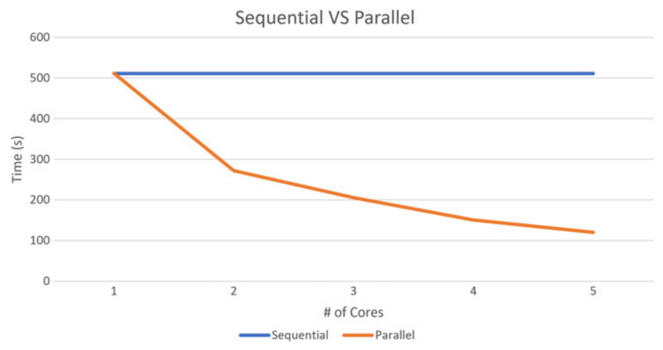

Figure 2 Time consumption vs the number of cores

TABLE 1 Time consumption vs the number of cores

Cores #	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

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

Listing 6 A source program example

3 | TASK PARALLELIZATION

Another problem is multi-task parallelization. A simple example is displayed in Listing 6. Two variables a and b store the returned values from two functions and they are then accessed in the flowing 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.

Being processed on 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 one is converting the synchronous program to asynchronous mode, and the other one is processing the task on the Spark platform.

The proposed design of this type of translation is represented in Figure 3. 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.

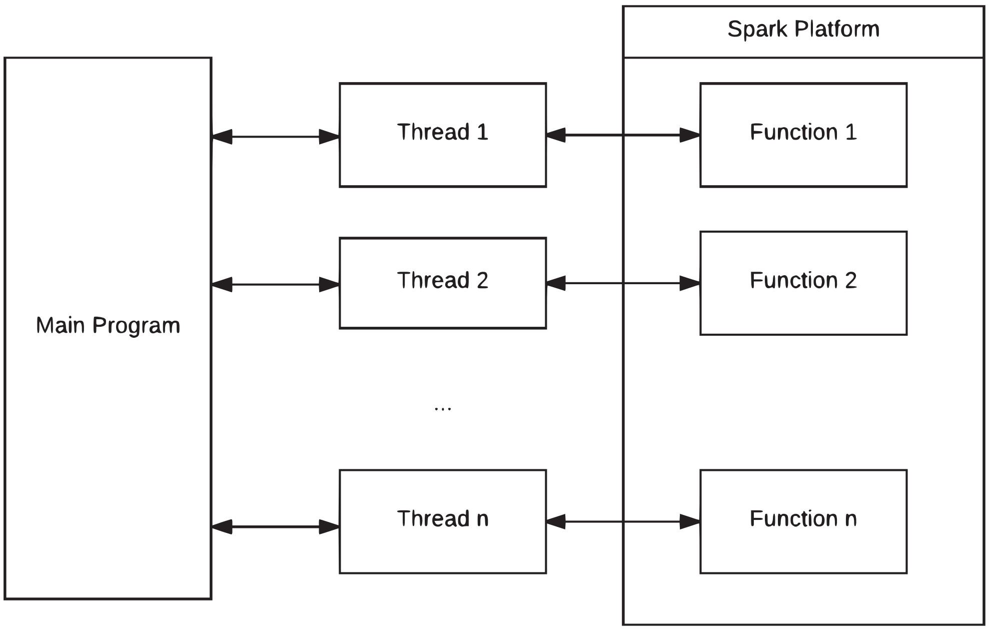

Figure 3 The structure of the translation resulting program

```
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();
```

Listing 7 A translation resulting program example

3.1 | Synchronous to asynchronous

Under the synchronous mode, execution of instruction should wait till the previous instruction is terminated in a regular Java program. Although this mode is easy to be controlled and understood, there exists a problem. For instance, in Listing 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 Listing 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.

The code in Listing 7 is the translation result of the program in Listing 6. Each source instruction in Listing 6 is translated to three statements. These statements are divided into two groups and located in two different places. One group is at the original locations of the source code, and the other one 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 constants, temporary variables are required to store them. After that, the original function is simply placed in a new thread with the constant parameters.

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

```
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;
}
```

Listing 8 A directive example

```
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);
});
```

Listing 9 An async spark program example

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

Similar to 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.

There are two types of directives 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 is crucial to notice that these two types of directives should work in pairs. In the proposed design, the size of the thread pool is set according to the number of the targeted instructions automatically.

3.2 | Process single function on Spark

The second step in the Task-Parallelization translation is to process these instructions on the Spark platform. The major problem in this step is that the Spark platform is not designed for single function processing. In general, the RDD contains a large number of data, and it processes them in parallel. However, 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.

The statements in Listing 9 describes how to process a single function on the Spark platform. It replaces the thread instructions in Listing 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. In this way, the original functions can be processed on the Spark platform.

3.3 | Evaluation

To evaluate the performance of the Task-Parallelization translation, the next experiment uses the same computer cluster like 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 can hardly benefit from the growing amount 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 amount of calculations. There are five asynchronous functions; each performs a different number of floating-point calculations from 1 billion to 5 billion. Table 2 and Figure 4 describe the time consumption of all tests.

In Figure 4, the blue and the orange lines represent the execution times of 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 cost because it adds one more logical layer. The two overhead costs, one from Java thread pool and the other one from the Spark platform, are almost constant as long as the size of thread is fixed. In conclusion, the speed-up can be larger in the bigger data size.

TABLE 2 Time consumption vs the size of calculation

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

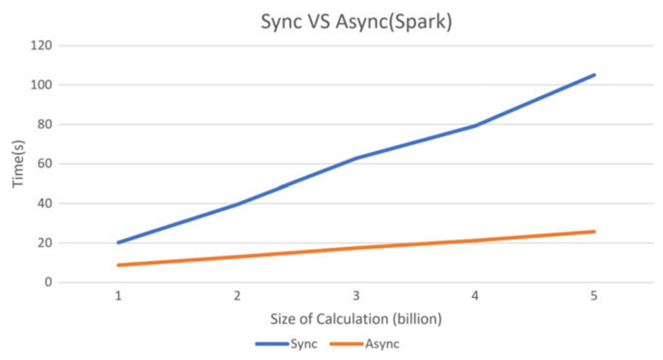

Figure 4 Time consumption vs the size of calculation

4 | FOR-LOOP IN TASK PARALLELIZATION

Generally speaking, functions with iteration or recursion consume the major resource in a single program. Normally, major cloud computing platforms are not designed for recursion, and it is hard to be parallelized due to data dependency. On the other hand, iteration is much easier to be parallelized if there is no data dependency, such as the situation in the Loop-Parallelization translation.

Discovered during the translation research, a new question is that can the Loop-Parallelization and the Task-Parallelization problem be combined together? If asynchronous functions in the Task-Parallelization translation contain larger size iterations, can the time consumptions be reduced by parallelizing these iterations? To solve this problem, a combination translation is discussed in this chapter.

Figure 5 describes the design of the Loop-in-Task translation. In short, this type of translation combines the previous two types together. 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 this 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 in another one, only the for-loop part of the asynchronous function, not the entire function, can be processed on the Spark platform. In fact, as mentioned before, this type of translation focuses on the functions that consume most resources on the iteration part. Processing for-loops in parallel on the Spark platform can significantly reduce the time cost of such program. If the program being translated does not contain a large loop, using the Task-Parallelization method is a better solution.

Listing 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 Listing 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 required to be processed in asynchronous mode.

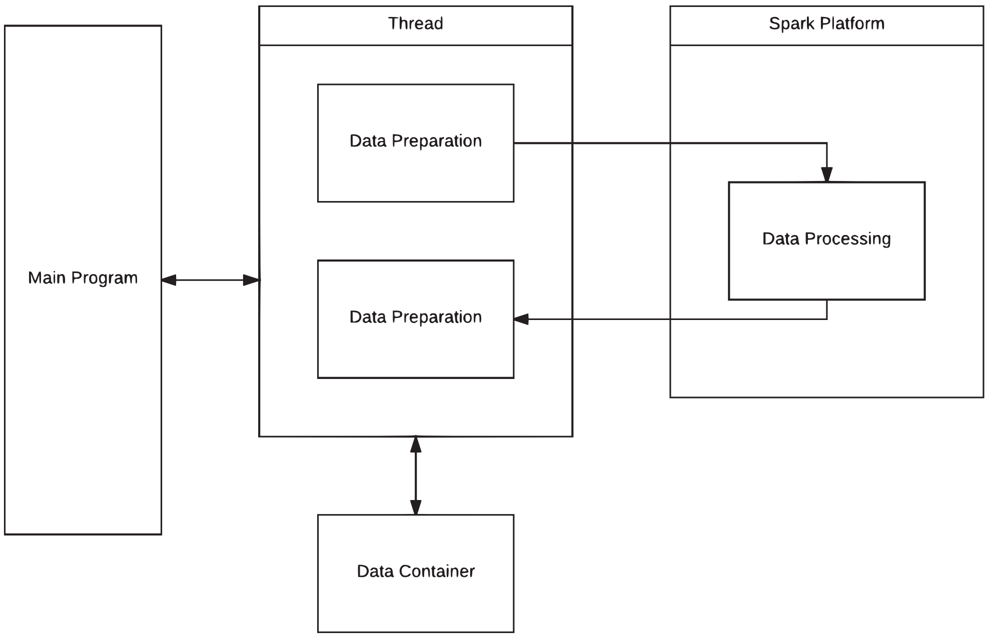

Figure 5 The structure of the translation resulting program

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

Listing 10 A source program example

```
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));
    ...
}
```

Listing 11 A spark configuration example

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 exactly the same as the codes described in the first step of the Task-Parallelization translation in Listing 7. The second part is the side functions, which are the *f1* and *f2* in this example. Their result codes are similar to the codes in the Loop-Parallelization translation. Yet the Spark configuration should be moved to the main function of this program because all side functions share the same configuration.

When Spark configuration statements are inserted, a problem is that 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 Listing 11. A new reference parameter *sc* is added to the *func* function's reference.

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

4.1 | Evaluation

The evaluation method for the Loop-in-Task translation is more complicated than that of the previous two chapters. The performances of the translation resulting program 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 previous two approaches. This experiment is also implemented on the same computer cluster as the previous ones.

The first experiment compares the performance of the original program and the resulting program on different calculation sizes. The experiment results are described in Figure 6 and Table 3. This test is similar to the one in the Task-Parallelization chapter. However, as shown in the Table 3, the speed-up increment is smaller than the one in that chapter because the overhead cost of the Loop-in-Task translation is larger.

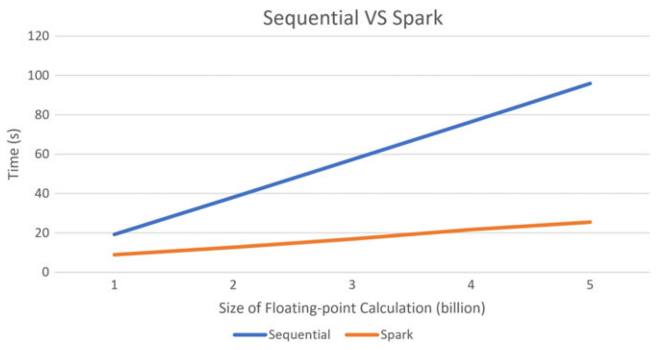

Figure 6 Time consumption vs the size of calculation

TABLE 3 Time consumption vs the size of calculation

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 4 Time consumption vs the number of cores

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

The maximum speed-up in this configuration is five, same as the amount of the cores. And the speed-up will get closer to five as the size of the calculation grows. This conclusion is based on an evidence shown through the comparison between the first test in Table 4 and all the results in Table 3. The speed-up in the first test in Table 4 is larger than all of those in Table 3 when the size of the calculation in the former is 10 billion, largest among all.

The second test is to fix the calculation size to 10 billion and manipulate the number of computing cores. Figure 7 shows that the Spark program benefits from the growth of the core number in all five tests. Meanwhile, the increment of speed-up is reduced along with the growth of cores, as shown in Table 4, because the calculation size is not large enough. When processing time is limited, the overhead cost takes more proportion. Since the overhead cost is almost fixed, it will take more proportion of the total time consumption when the latter is lower. Furthermore, the increment of speed-up 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.

5 | CONCLUSION

The J2S translator proposed in this paper is a new generation of the J2M translator. These two translators have the same framework design, but the J2S translator has more functionalities, which can translate more types of the source program and is built on the in-memory model. The evaluations

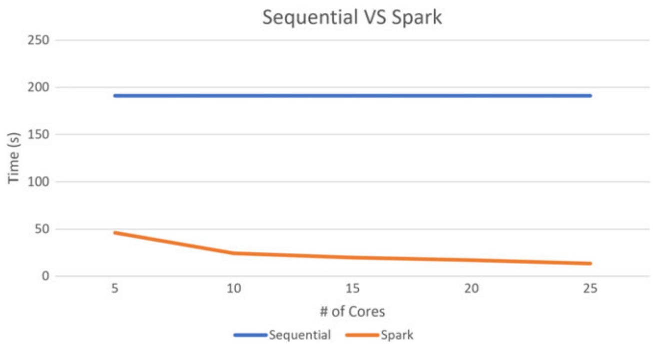

Figure 7 Time consumption vs the number of cores

of three types of translation demonstrate that all of the three translation results work well in their domains. Meanwhile, the J2S translator is more productive because it does not require human participation during the work procedure of the translation resulting program.

In the current version, the J2S translator still has some disadvantages. For example, it packages all intermediate data and sends them to the cloud platform including some useless data. It will also transmit redundant data after it has been used for multiple times. The performance of the resulting program can be significantly improved if these problems are solved. As another example, J2S translator can only translate Java program at the current stage. However, besides Java, there are many popular programming languages in the big data processing area. Implementing the automatic translation method on these languages can create powerful and efficient tools for data engineers. The J2S translator only provides preliminary ideas of the automatic translation from the sequential to parallel programs on the cloud platform, and we hope the related projects in the future can get inspirations from ours.

ORCID

Bing Li http://orcid.org/0000-0002-8233-5191

REFERENCES

1. Armbrust M, Fox A, Griffith R, et al. A view of cloud computing. *Commun ACM*. 2010;53(4):50-58.
2. Armbrust M, Fox A, Griffith R, et al. Above the Clouds: A Berkeley View of Cloud Computing [Technical Report]. Berkeley, CA: University of California; 2009.
3. Jackson KR, Ramakrishnan L, Muriki K, et al. Performance analysis of high performance computing applications on the Amazon Web Services cloud. Paper presented at: 2010 IEEE Second International Conference on Cloud Computing Technology and Science; 2010; Indianapolis, IN.
4. Dikaiakos MD, Katsaros D, Mehra P, Pallis G, Vakali A. Cloud computing: distributed internet computing for IT and scientific research. *IEEE Internet Comput*. 2009;13(5):10-13.
5. Buyya R, Yeo CS, Venugopal S. Market-oriented cloud computing: Vision, hype, and reality for delivering IT services as computing utilities. Paper presented at: 2008 10th IEEE International Conference on High Performance Computing and Communications; 2008; Dalian, China.
6. Pan Yi, Zhang J. Parallel programming on cloud computing platforms. *J Converg*. 2012;3(4):23-28.
7. Thusoo A, Sarma JS, Jain N, et al. Hive: a warehousing solution over a map-reduce framework. *Proc VLDB Endowment*. 2009;2(2):1626-1629.
8. White T. *Hadoop: The Definitive Guide*. Sebastopol, CA: O'Reilly Media, Inc; 2012.
9. Lee R, Luo T, Huai Y, Wang F, He Y, Zhang X. YSmart: Yet another SQL-to-MapReduce translator. Paper presented at: 2011 31st International Conference on Distributed Computing Systems; 2011; Minneapolis, MN.
10. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107-113.
11. Zhang J, Xiang D, Li T, Pan Y. M2M: a simple Matlab-to-MapReduce translator for cloud computing. *Tsinghua Sci Technol*. 2013;18(1):1-9.
12. Li B, Zhang J, Yu N, Pan Y. J2M: a Java to MapReduce translator for cloud computing. *J Supercomput*. 2016;72(5):1928-1945.
13. Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Paper presented at: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation; 2012; San Jose, CA.

How to cite this article: Li B, Xiao X, Pan Y. Automatic translation from Java to Spark. *Concurrency Computat Pract Exper*. 2018;30:e4459. https://doi.org/10.1002/cpe.4459