Refactoring Java Code to MapReduce Framework

Rui Feng
Inner Mongolia University
Hohhot,China
15506597902@163.com

Junfeng Zhao*
Inner Mongolia University
Hohhot,China
cszjf@imu.edu.cn

Abstract—**Cloud computing has evolved into an infrastructure tool for scientific research and computing application. For many enterprises, it has become a trend to migrate their applications from the local to cloud. To leverage cloud computing infrastructure, some legacy code for special business process need to be refactored to the programming models of cloud computing. The desired approach is to design an automatic tool for refactoring legacy code to the target code that can execute on cloud computing platform effectively. In this paper a new approach is proposed, which can automatically refactor Java sequential program into MapReduce paradigm. The approach works by first translating input code into functional representation, with loops succinctly encapsulated by fold operations. Then, guided by the transforming rules, the approach generates equivalent MapReduce programs. The rules use group-by operations to enable greater parallelism. Finally, a series of mapping rules is applied to map immediate code to the target code running on Spark. A new tool designed using this approach, JMRT, was evaluated using real world benchmarks. The experimental results show that the approach can generate the desired MapReduce program and the business execution efficiency can be improved.**

Keywords-Automatic Refactoring;MapReduce;Fold Operation；Parallelism;Program Transformation

I. INTRODUCTION

In the past decade, MapReduce [1] has attracted interest as a parallel programming model, independent of difficulties of distributed computation [2]. Main-stream MapReduce frameworks equip average developers with the tools that can instantly transform them into distributed system developers [3,4,5]. Specifically, we can select the appropriate variation by looking at the type information of the λm function used by map. The tool provides developers with abstract data-parallel operators map and reduce that shield them from the complexity of distributed computing. Therefore, the use of cloud platforms is increasing rapidly. In addition to developing new applications directly on the cloud, more and more applications are being migrated from local servers to cloud servers.

There is a problem in the process, how to reduce the migration cost. To maximize the performance of cloud applications, the code migration method needs to transform their programming model from sequential to parallel. 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 legacy code due to data dependency and the costs. Therefore, it is necessary to find a way to automatically translate sequential code into executable code under the MapReduce programming model.

DOI 10.18293/SEKE2021-059

In the current research, most of the work aim to refactor the code using traditional parallel transformation methods. A common method is to obtain the code data access pattern and determine the order of retrieving data values during the program runtime, and then use the data reordering method to complete code transformation [6]. However, this method is not suitable for the transformation of Java code to MapReduce programming model, and many extensions to this method cannot effectively solve this problem. In addition, there is work aimed at refactoring code using design code templates. Paper [7] classifies the source code according to business logic, and then proposes corresponding reconstruction rules for each type. The approach is limited to specific access patterns, without considering all the code scenarios, it is necessary to propose a more comprehensive method to achieve effective reconstruction of Java code. To sum up, the existing refactoring methods are not perfect, and the refactoring tools are not mature enough. It is necessary to propose a more comprehensive method to achieve effective reconstruction of Java code in cloud migration.

In this paper, a comprehensive automatic translation method for sequential code to MapReduce programming model is proposed, which can effectively implement Java code to MapReduce code refactoring. A tool that implements our method is generated, which can handle complex input programs.

Fig. 1 illustrates the design of our method, which translates sequential code into equivalent MapReduce programs. The method includes the following steps. The first step of the method is to translate the input program into a functional representation via GSA (Gated Single Assignment) form [8]. Although the functional form is semantically equivalent to the original imperative code, unfortunately exposes no parallelism. since a simple data-parallel program consists of at least a map followed by a reduce. To address these problems, the transforming rules are designed to govern where map functions can be introduced in a semantics-preserving manner. In more complex cases where loop iterations access overlapping locations, this method uses Spark's groupByKey operation to group operations by access, exposing more fine grained parallelism than the previous method. Final, a series of mapping rules are designed to map the executable MapReduce program to the target Spark platform.

The structure of the article is as follows. In the approach presented in this article, code under two different programming models needs to be transformed, an IR (intermediate representation) is needed to assist in the transformation. Therefore, the generation process of functional IR is described in the chapter 2. Since Java sequential code cannot be transformed directly to intermediate code, a representation that helps Java code to be transformed to IR is needed. In the first section of the second chapter, this form is briefly described; in the second section, the algorithm for translating sequential code to this form is shown; in the third section, the functional IR used in this paper is described; and in the fourth section, the rules for generating intermediate code are described. Once the functional IR has been got, the next step is to generate the executable code under the MapReduce programming model. Therefore, in chapter 3, we first introduce the transformation rules that introduce parallelism into intermediate code, and then introduce the mapping rules that generate executable code on the target platform. The refactoring tools and experimental validation are presented in chapter 4.

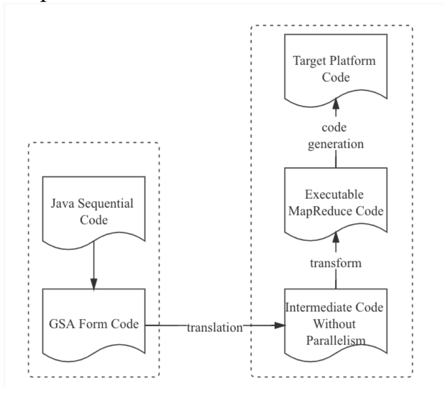

Figure 1. Overview of the method

II. GENERATING FUNCTIONAL IR

A. *Introduction of GSA Form*

Since the source code cannot be directly transformed to the IR, a form is needed to help the transformation. GSA form is a representation based on static single value assignment (SSA) [9,10], which is a representation generated by static code analysis [11, 12]. The GSA form assigns unique names to variables in the program and embeds the gated predicate information φ function, so as to realize the analysis of program's data flow and control dependency information. Specialized gating functions (γ, η and μ) are introduced in GSA to include the predicate of conditional branches. Different pseudo-functions replace the φ functions at different confluence nodes in the program control flow graph.

A program in GSA form is essentially a functional program, which facilitates program transformation. The functional intermediate form is derived from program in GSA form. GSA form is the basis for converting imperative loops to a MapReduce style.

B. *Construction of GSA Form*

An algorithm of translating source code to GSA form is proposed in this paper. Two steps are required to translate source program to GSA form. The first step is to insert the special assignment statement φ functions into certain places of the program and replace the φ function with μ, γ and η function. In the second step, each reference to v in the program is replaced by a reference to one of the new names v_i.

In order to transform sequential code to GSA form, the data structure CFG [13] and the dominance relation [14] between nodes exists in the CFG are used in this paper.

Dominance Frontiers are used to find where φ functions are needed. The dominance frontier DF(X) of a CFG node X is the set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y.

The method for placing φ functions is: Whenever node X contains a definition of some variable v, any node in the dominance frontier of X needs insert a φ function for v. The following code shows how to insert φ functions. The input of the algorithm is the CFG information of the source program. Then, the output of the algorithm is a CFG which contain the φ function. Several data structures are used: P is an array that stores the CFG nodes being processed. Process is an array of flags, one for each CFG node. When node X is currently added to P, the Process(X) is 1. DomFron is an array of flags, one for each node. When the φ function about variable v has been inserted in node X, DomFron(X) is 1.

```
Algorithm: Insert φ functions.
1   for each v do      // Assign each variable as follows
2      P←0
3      Process←0
4      DomFron←0
5      for each X ∈ S(v) do      //Iterate the node that contains the
    assignment statement for the variable v
6         Process (x) ←1
7         P←P∪{x}
8      end
9      while |P|>=1do
10       take (x,p)     //Fetch node X from the collection of nodes being
    processed
11       for each Y ∈ DF(X) do      //Iterates through the nodes in the
    DF set of node X
12          if DomFron (Y)=0
13             then do
14             add φ-function for v to Y      //Inserts the φ function of
    the variable v in the node Y
15             DomFron (Y) ← 1
16             if Process(Y)=0
17                then do
18                Process (Y) ← 1
19                P ← P∪{Y}
20             end
21          end
22       end
23    end
24  end
```

Use a more precise control predicate function in the code of the GSA form. The use of three functions improves the analysis accuracy of variables, and ultimately improves the accuracy of parallelization. The third step uses the following rules to replace the φ function in the program to complete the transformation of Java source code to GSA form.

μ function: Replaces those φ functions at the head of a loop. Each μ function combines the index value initialized by the loop with the index value calculated in the loop body. In the first iteration, the μ function returns the first argument i_0, which is the value assigned before entering the loop, otherwise, it returns the second argument i_1, which is the value from the previous iteration. It involves the change of loop index and places it in the loop header.

γ function: Replaces those φ functions located at the confluence nodes that have no incoming back edges. The back edge is the edge from the descendants to the ancestors in DT. The γ function selects the value of a variable computed by the if statements, and the condition in if statements as the parameter.

η function: Replaces φ functions at the nodes that contain loop exit edges as incoming edges. It selects the last value at the end of the loop. The function is placed where the loop exits. The

GSA form is derived from the CFG of the source program, so the semantic equivalence is verified.

The following code shows the construction of the GSA form. S is a stack that stores every write operation of variable v, one for each variable. Count is an integer, one for each variable. The Count value of any v represents how many assignments to v have been processed. The algorithm uses a depth-first search method to access the Dominator Tree (DT)， which contains information about variables and is directly generated by existing algorithms, and the search starts with Entry, where the entry value of v is represented by an empty assignment on the right. In DT, the children of a node X are all immediately dominated by X. After renaming each variable v to v_i, the search will continue to other nodes. When the statement of each node on the variable v is processed, the stack of the current variable v is cleared. The words predecessor and successor refer to CFG. The words parent, child, ancestor, descendant refer to the DT.

```
Algorithm: Construct GSA form.
Count←0
S is empty
call Visit (Entry)      //Start the search with the entry node
Visit(X):      //When you search for node X
   for each assignment A in X do       //Do the following for the assignment statement in the node
      for each variable v in RHS(A)
         rename v with $v_i$ where i = top(S(v))
      end
      for each variable v in LHS(A)
        i ← the Count value of v
        rename v with $v_i$ in A
        push (S, i)
        the Count value of v ← i+1
      end
   end
   for each φ function in X do      //Rename a variable in a function within a node
         rename j-th variable v in φ with $v_i$ where
         i = top(S(v))
   end
   for each Y∈ children (X) do       //Continue to call the children of the current node
         call Visit(Y)
end
```

C. Functional IR

In current work, the intermediate representation is typed lambda calculus. Lambda calculus is the basic mathematical theory of functional programming language [15]. It can describe and analyze programming language and use λ expression to represent programming language. The current IR is a typed functional language based on lambda calculus. Typed lambda calculus assigns each item in the lambda calculus a type, which can be int, string, etc. A simple example: if the variable x has type σ and there is an expression M, then λ x:σ.M defines a function that maps any x in σ to the value given by M. For translating an entire program block to λ expression, a reasonable typed grammar is let x = M in N, which means to constrain x to M within N. In other words, the value of let x = M in N is the value obtained by setting x to M in N. a[b] and a [b: =c] are read and write access at index b of a map(array) a. Table 1 lists the set of operations in our IR. These expressions are well-known.

TABLE 1.IR

IR operations	Description
var a | b |c |… | <var,var,…>	variable declaration
Exp λ Var[:Type].Exp| let Var=Exp in Exp | Exp Exp | <Exp,Exp,…> | Exp[Exp] | Exp[Exp:=Exp]	expression declaration
Type A | B | C|…| Type ⟶Type| <Type,Type,…> | Type[Type]	type declaration

Figure 2 shows the data structure in the intermediate code and the functions operating on it. Most of these structures are well known, and they are described as needed in the article.

(data structures)

$M[A]$: multiset with values of type A

$M[K,V]$：map with keys of type K and values of type V

(functions)

$$map: (A \to B) \to (M[A] \to M[B])$$

$$map: (< K,V > \to W) \to (M[K,V] \to M[K,W])$$

$$groupByKey: (A \to K) \to (M[A] \to M[K,M[A]])$$

Figure 2. Built-in data structures and functions.

D.Transform GSA Form to Functional IR

New transformation rules are proposed to transform GSA form code into an IR. Transformation rules transform loop and non-loop statements in code, respectively.

1) Non-loop Rules

The method transforms GSA form to the functional IR by applying the rules in Fig. 3. Simple rules are discussed first. Each non-GSA assignment statement is directly converted to the corresponding *let ... in ...* statement. Any branch instruction is skipped, left to be handled when reaching its associated node. The return instruction is replaced with the returned variable, which eventually sits at the innermost level of the let nest. Transform each γ instructions into a functional *if* statement whose condition comes from a branching instruction, and the branches being the arguments of the γ instruction. As the instructions are visited in topological order, the variables holding the result for each of the two branches are already available in scope.

$$x = E \prec R \to let\ x = E\ in\ R$$

$$a[x{:=}\ y] \to a[x{:=}\ y]$$

$$return\ \ x \to x$$

$$x = \eta(C, x_0, x_1) \prec R \to let\ x = if\ C\ then\ x_0\ else\ x_1\ in\ R$$

Figure 3. Non-loop Rules.

2) Loop Rules

The transformation of loops is essential to introduce parallelism, so it is very important to translate the loop structure. Fold operation is introduced to solve this problem. In functional programming, fold is a standard operator that encapsulates a pattern of function for processing recursive calls. Moreover, the fold operator is equipped with a proof principle called universality, which provides a mathematical principle for solving rule proofs. Fold has been introduced to ensure the correctness and availability of loop transformations.

$$\begin{aligned} &for(i = \varphi(i', i''), i < l, E\{x_1 = \varphi(x_1', x_1'')\ldots\}) \to \\ &\quad let\ f = \lambda\, x_1, x_2, \ldots, x_n.E\ in \\ &\quad\quad let\ \ x_n = fold^{T(x_1', x_2', \ldots, x_n')}(f)\ in\ range(i, l) \end{aligned}$$

Figure 4. Loop Rules

The more complex rule translates loops to applications of the fold operator. In Fig.4, a loop is specified in terms of its φ variables, which include the index variable i and other variables x_1, x_2 ,updated in the loop body. These variables characterize all possible effects of the loop visible to subsequent code. For each φ variable x_n , we use $x_n{}'$ to refer to the value coming from outside the loop, and $x_n{}''$ for the new value produced by the loop. The loop gets translated to a fold over the domain of values for the index variable, from i to l. The function f takes as arguments the current x_n values, runs the body of the loop E once for those values, and returns the new $x_n$values. The initial value for the fold is a tuple of the $x_n{}'$ values coming from outside the loop.

To obtain the fold operation over different types of input variables, the function E must first be obtained. The code is then transformed using the definition of the fold operation.

The definition of the functor of the following expression is used to construct the function E. Functor E associated with each constructor $\mathrm{C}_i\colon(\mathrm{t}_{i,1},\ldots,\mathrm{t}_{i,m_i}) \to \mathrm{T}(\alpha_1,\ldots,\alpha_p)$ is a (p+1)-adic function (Where p is the number of universally quantified type variables in the left-hand side of T's type equation):

$$\mathrm{E}_i^T(f_1, \ldots, f_T) = \lambda(x_{i,1},\ldots,x_{i,m_i}).\,(k[t_{i,1}]x_{i,1},\ldots,k[t_{i,m_i}]x_{i,m_i})$$

where the bound variable x_{i_j} has type, t_{i_j} and $k\left[t_{i,j}\right]$ represents a function that can be obtained by the following rules:

$$K\left[T\left(\alpha_1, \ldots, \alpha_p\right)\right] = f_T$$
$$k[t_1, t_2] = \lambda\, x_1, x_2.\,(k[t_1]x_1, k[t_2]x_2)$$
$$[u \to v] = \lambda\, h.\, k[u] \circ k[v] \circ h$$

Now, it is possible to describe the fold operator for any loop with expression and functions. The fold function over T $(\alpha_1 \ldots \alpha_p)$ is defined by the following equations.

$$fold^T(\overline{f}) = fold^T(E_i^T(f_1,\ldots,f_n),\ldots)$$

where $\overline{f} = (f_1,\ldots,f_n)$. Each f_i in $\overline{f}$ is a function.

The functional intermediate form is semantically equivalent to the original imperative code but unfortunately exposes no parallelism, since the loop operation is still sequential.

III. GENERATING EXCUTABLE MAPREDUCE CODE

A. *Parallelize transformation rules*

In the previous chapter, a functional IR with fold operations that is semantically equivalent to sequential code is generated. However, the current IR is still sequential and does not show any parallelism. Therefore, the parallelization transformation rule shown in Fi.6 is proposed to reveal the parallelism.

(extract map from fold)

$$\frac{fold^{\mathrm{T}(x_1^0,\ldots,x_n^0)}\lambda\, T(x_0,\ldots,x_n).\,E^T}{\left(fold^{\mathrm{T}(x_1^0,\ldots,x_n^0)}\lambda\, T_n(x_0,\ldots,x_n).\,f_n^{T_n}\right) \circ \left(map\,\lambda\, T_c(x_0,\ldots x_n).\,f_c^{T_c}\right)}$$

(extract groupByKey)

$$\frac{fold^{r_0}\lambda\, r\, v.\, r[E := B]}{(map\,\lambda\, k\, l.\,(fold^{r_0[k]}\,\lambda\, g\, v.\, C)l) \circ (groupByKey\,\lambda\, v.\, E)}$$

Figure 5. Rules Revealing parallelism.

The transformation that reveals parallelism is the "extract map from fold" rule in Fig.5. It transforms a fold by extracting independent computations from the function f and transform it into a (parallelizable) map operation. An independent function must make no reference to an accumulator parameter. For example, $\mathrm{fold}^{\mathrm{T}}(\lambda\, \mathrm{r\, k\, v.\, r, f\, k\, v})$ is transformed to $\mathrm{fold}^{\mathrm{T}}(\lambda\, \mathrm{r\, k\, v_f.\, r + v_f}) \circ$ $(\mathrm{map}\,\lambda\, \mathrm{k\, v.\, f\, k\, v})$, as $(\mathrm{f\, k\, v})$ is independent. After the transformation, the purely functional map can be easily parallelized.

The "extract map from fold" rule, shown in Fig.5, matches on any fold taking any type variables and functions. The fold operation E is split into the composition of functions $f_n \circ f_c$. such that f_c is independent of other "iterations" of the fold's execution. If the fold is seen as a loop, f_c does not have any loop carried dependencies. f_c is pulled out into a map.

While the "extract map from fold" rule exposes significant parallelism, it cannot handle the situation when distinct loop iterations can update the same array location. MapReduce applications like word count mentioned earlier often work around such issues by using a shuffle operation to group inputs by some key and then process each group in parallel.

The transformation used for grouping by word is an application of the "extract groupByKey" rule shown in Fig. 5. The groupByKey operation clusters the elements of a collection of type M[A] according to the result of the function A to K. It returns a map from keys K to lists M [A] of elements in the original collection that map to a specific key. The rule matches any fold with a body which is an update of a collection at an index E.

The output code first groups the elements of the collection by the index expression (groupBy λ v. E), and then it folds each of the groups using the update expression B from original body of the loop. groupByKey's output is a Map from each distinct value of E to the corresponding subset of the input collection. The map operation's parameters are k, which bounds to the keys of the grouped collection), and l which contains subset of the input collection. The fold starts from the k value of $\mathrm{r_i}$, and folds l using the operation C, which is original expression B with accesses to index E of the old reducer replaced with g, the new parameter corresponding only to the k-index of r.

B. *Object Code Generation*

When the program is translated, IR with MapReduce programming logic is generated. The mapping of the intermediate code to the target platform only involves the transformation of syntax in different programming languages and does not involve the conversion of semantics. Therefore, the corresponding mapping rules are constructed according to the syntax structure of the target platform and API calls. We list a subset of such mapping rules for the Spark RDD API [16].

map(Input,λ_m:T→list(Pair)) = Input.flatToPair([[λ_m]]);
map(Input,λ_m:T→list(U)) = Input.flatMap([[λ_m]]);
map(Input,λ_m:T→U) = Input.map([[λ_m]]);
reduce(Input:list(U),λ_r) = Input.reduce([[λ_r]]);
reduce(Input:list(Pair),λ_r) = Input.reduceByKey([[λ_r]]);

An expression in the IR language is used as input, and the output is an equivalent expression in Spark. Since Spark provides multiple variations for the operators defined in our IR, such as a map, we can select the appropriate variation by looking at the type information of the λm function used by the map. For example, if λm returns a list of Pairs, we translate to JavaRDD.flatMapToPair.

IV. REFACTORING TOOL AND EXPERIMENTAL EVALUATION

A. *Refactoring Tool*

Although there are many loops in legacy code, not all of them can be refactored in parallel based on the MapReduce model. Another work by our research group has proposed a way to identify parallelizable loops and annotate them with specific parallel tags [17]. Based on the refactoring method proposed in this paper, a tool named JMRT supporting refactoring is designed. The tool obtains the code in the program according to the parallel mark, and then uses the components designed by the above method to refactor the code. The working process of the tool is shown in Fig.6.

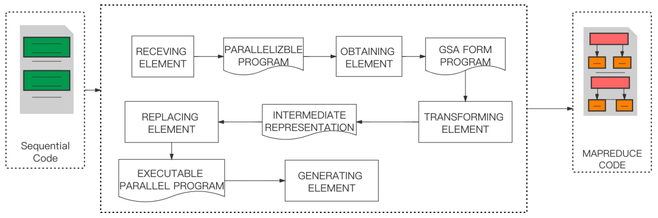

Figure 6. Working process of JMRT.

As seen in the picture, the components of JMRT comprise: receiving processor receives the sequential program that needs to be process; obtaining processor obtains the GSA representation of the source code; transforming processor translates the GSA representation to lambda representation; replacing processor replaces the loop with rules to generate executable parallel programs; and generating processor generates the executable MapReduce program on spark. In one example, any steps may be carried out in the order or the steps may be carried out in another order.

B. *Experimental Verification and Result Analysis*

1) *Refactoring Experiment*

In this section, a benchmark is used to test the feasibility of JMRT. Phoenix [18] is a standard MapReduce benchmark suite that provides both MapReduce and corresponding sequential implementations. It contains the main calculations from the application domain, such as enterprise computing (Word Count, Reverse Index, String Match), scientific computing (Matrix Multiply), artificial intelligence (KMeans, PCA, Linear Regression), and image processing (Histogram). Table 2 shows the number of loops and loop nests in the original programs, and whether the translation is successful or not.

TABLE 2. Program information and results.

Program	Loop nests	Loops	Translation
Word Count	1	1	√
Histogram	1	1	√
String Match	1	1	√
Linear Regression	1	1	√
Matrix Product	3	2	√
PCA	2	5	√
KMeans	2	6	√

A concrete example is used to demonstrates how JMRT translates sequential code into MapReduce programs. Fig. 7 shows the sequential Java code, our starting point. The program iterates through a list of documents inputs, accumulating the word counts into the map.

```
static Map<String, Integer> wordCount(List<String> inputs) {
     Map<String, Integer> map = new HashMap< > ();
    for (int i=0; i<inputs.size (); i++) {
        String [] inputSplit = inputs.get(i). split (" ");
        for (int j=0; j<inputSplit.length; j++) {
            String word = inputSplit[j];
            System.out.printLn("value"+word);
            Integer value = map.get(word);
            System.out.printLn("key"+map.get(word));
            if (value == null)
                value = 0;
            map.put (word, value+1);
            System.out.printLn(map.values ());}
        }
        return map;
    }
```

Figure 7. Java sequential code.

The section “construction of GSA form” explains how translate the inner loop part of input program into a GSA form. Fig. 8 gives the obtained GSA form using the algorithm.

```
$j_0 = 0$
**do** $j_1 = \mu\ (j_0, j_2)$ ,inputSplit.length
    word = inputSplit[$j_2$]
    $value_0 = map_0$[word]
    $value_2 = \gamma\ (value_0 == null, value_0, value_1)$
    count = $value_2 + 1$
    $map_1 = map_0$[word: = count]
    $j_2 = j_1 + 1$
**enddo**
    $j3 = \eta\ (j0, j2)$
    $map_2 = \eta\ (map_1, map_0)$
```

Figure 8.GSA form for the inner loop of the code.

Then the program is translated into the functional IR. Thus, the code in Fig. 8 is converted to:

```
fold^m ({λ m word.
      let value = map[word] in
         map [word: = (if value == null then 0 else value) + 1]
      }) inputSplit
```

JMRT generates a solution for parallelization. Rather than directly avoiding non-linear variables in the code, the program can examine them to reveal parallelism. Therefore, use the parallelization rule to transform the above example into code like the following:

map (λ k l. $fold^m$(λ g w.g+1)l) ∘(groupByKey λ word.word)

Finally,the tool generates the code as follows:

. λ_{map} ({(i, inputs) => inputs.Split (" ")})

. λ_{map} ({(key, value) => (key, 1)})

. λ_{reduce} ({($value_0$, $value_1$) => $value_0 + value_1$})

Each document is divided into multiple words, and then they are processed into (word, 1) pairs using the map function. The reduce function groups "1" values by their key, and then reduces the grouping by the add operation, thereby effectively counting the number of words. In this way, it reaches a form similar to the traditional MapReduce solution for the WordCount problem. The documents are split into words, which are then shuffled and the numbers of elements in each word package is counted. Finally, the corresponding Spark code is shown as follws.

```
val wordCount = inputs.split(inputs=>inputs. Split (" "))
val pairs =wordCount.map (word=> (word, 1))
val results = pairs.reduceByKey((value1,value2)=>value1+ value2)
```

2) *Result Analysis*

In terms of the experimental environment, the experiments were run on a quad-core Intel i7 at 2.6GHz with 16GB of RAM. One of the defining characteristics of translation results is performance, especially execution speed. We create a performance test experiment to test the performance of this translation method.

Fig.9 compares the execution time of the sequential code and refactored code on Spark. The Spark translations the tool generated for this benchmark performed 10.9× faster on average than the sequential versions. When the input data set is small, the execution efficiency of sequential code is better than the executable code under the reconstructed MapReduce programming model. When the input data set is gradually enlarged, the advantages of the reconstructed code are gradually highlighted, and the corresponding execution efficiency is also continuously improved. This is because it takes a certain amount of time for each cluster to start and load data. When the input data set is small and there are many nodes, the time overhead brought by communication far exceeds the time advantage brought by parallel computing. Therefore, the execution efficiency of sequential code is better than that of the reconfigured code. However, the size of the input data set when the two types of code execution efficiency are demerged is not measured in this article, because it depends on many factors such as the network of the cloud platform.

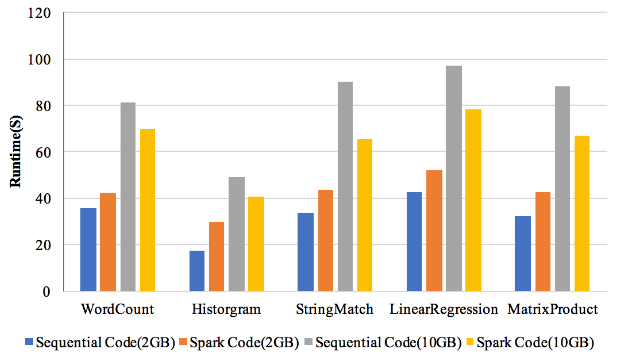

Figure 9. Runtime comparison

V. Related Work

Source-to-Source Compilers. MOLD [19] is a compiler that relies on syntax-directed translation rules to translate Java programs into executable code under the cloud computing programming model.

Program Refactoring. M2M (Matlab-to-MapReduce) [20] is a refactoring tool for scripting languages. This tool can be used for basic numerical calculations. It can translate Matlab code to MapReduce code in a short time, far exceeding programmer Efficiency of hand coding. YSmart [21] is also a scripting language refactoring tool that can provide a general framework to transform complex SQL queries into optimized MapReduce jobs and efficiently execute them in distributed cluster systems. J2M [7] is a refactoring tool for programming languages. It needs to design function templates related to MapReduce, fill the template with code fragments extracted from the source program, and complete the conversion. In [22], a refactoring method is proposed, which divides the types of businesses that can be processed by MapReduce, and defines corresponding refactoring rules for different business types. This method realizes the automatic generation of target code.

VI. Conclusion

This article describes a method for translating sequential Java code snippets into executable code under the MapReduce framework. Code snippets are transformed by defining a transformation rule in the method, which introduces the map and groupByKey functions to introduce parallelism. Our experiments show that JMRT can transform benchmarks in real-world applications. The generated code executes faster than the original code and is competitive with handwritten code.

Acknowledgment

This work was supported by National Natural Science Foundation of China (No.61962039) and Inner Mongolia Natural Science Foundation of China (No.2019MS0632).

References

[1] Ieffrey Dean and Sanjay Ghemawat. “MapRedcue: simplified data processing on large clusters”, in OSDI,2008, pp.107-113.

[2] Smith C, Albarghouthi A. “MapReduce program synthesis”,Acm Sigplan Notices,vol.51,no.6,pp. 326-340,2016.

[3] Apache Hadoop 2021. http://hadoop.apache.org, last accessed 2021/03/11.

[4] Apache Spark 2021. https://spark.apache.org, last accessed 2021/03/11.

[5] Yuan Yu, Michael Isard, Dennis Fetterly,et al. “Dryadlinq: A system forgeneral-purpose distributed data-parallel computing using a high-levellanguage”, in OSDI, 2009,pp.1-14.

[6] M. Ravishankar, J. Eisenlohr, etc. “Code generation for parallel execution of a class of irregular loops on distributed memory systems”,in SC ’12, 2012,pp.1–11.

[7] B.Li,J.B.Zhang,N.Yu,etc. “J2M:a Java to MapReduce translator for cloud computing”,SuperComputing,vol.72,no.5,pp. 1928–1945,2016.

[8] Peng Tu and David Padua. “Gated SSA-based demand-driven symbolic analysis for parallelizing compliers”, in Proceedings of the 9th International Conference on Supercomputing, 1995, pp.414-423.

[9] L.R, J.B. “SSA-based MATLAB-to-C compilation and optimization”, in SIGPLAN, 2016 pp. 55-62.

[10] S.J, P. H. “Constructing HPSSA over SSA” , in SCOPES,2017, pp. 31-40.

[11] Louridas P. “Static code analysis”, IEEE Software, vol. 23, no.4,pp. 58-61,2006.

[12] Ferrante J, Ottenstein K J, Warren J D. “The program dependence graph and its use in optimization”, ACM Transactions on Programming Languages and Systems , vol.9,no.3,pp. 319-349,1987.

[13] Orailoglu, Alex, and Daniel D. Gajski. “Flow graph representation” , in Proceedings of the 23rd ACM/IEEE Design Automation Conference. 1986,pp.503-509.

[14] R. E. Tarjan.“Finding dominators in directed graphs” , SIAM J. Computing, vol.3,no.1,pp.62-89,1974.

[15] Sebesta, Robert W. Concepts of programming languages. Pearson Education, 2012.

[16] Maaz Bin Safeer Ahmd, Alvin Cheung. “Automatically leveraging MapReduce framework for data-Intensive applications”, in SIGMOD, 2018,pp. 1205-1220.

[17] ZhiMei Zhao.Distributed Parallel Analysis of Legacy Code[D].Inner Mongolia University, 2019.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. “Evaluating MapReduce for multi-core and multiprocessor systems”,in HPCA ’07, 2007,pp. 13–24.

[19] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. “Translating imperative code to MapReduce” in OOPSLA’ 14,2014, pp.909-927.

[20] Zhang J, Xiang D, Li T, “M2M: a simple Matlab-to-MapReduce translator for cloud computing”, Tsinghua Science and Technology, vol. 18, no. 1, pp. 1-9, 2013.

[21] Lee R, Luo T, Huai Y, “Ysmart: yet another SQL-to-MapReduce translator,” in ICDCS, 2011, pp. 25-36.

[22] J.F.Zhao and W.M.Wang, “Creative combination of legacy system and MapReduce in cloud migration”, International Journal of Performability Engineering, vol. 15, no. 2, pp. 579-590, 2019