Parallel Clone Code Detector in MapReduce[1]

Lin Ye
College of Information Science and Technology, Jinan University, 510632, Guangzhou, China
Email:ye_lin@126.com

Guoxiang Yao[2]
College of Information Science and Technology, Jinan University, 510632, Guangzhou, China
Email: yao@jnu.edu.cn

Abstract—Programmers often copy code to improve efficiency, and different developers may write the same code independently, these behaviors bring clone code to the project. Clone code makes the project hard to maintain and weakens the robustness, and the bugs in these code segments would undermine the whole project. The state-of-the-art clone code detectors are either not able to find code with same semantics, or computationally expensive. And if clone code detector is to be performed on plenty number of code, the main memory of one machine may not able to hold all the information.

In this paper we focus on the parallel of the clone code detector, we utilize the Program Dependence Graph (PDG)-based clone code detection method, which can not only check the code in contiguous syntax, but also the code with the same semantics. We present an approach to parallel the isomorphism matching in the PDG. By using MapReduce paradigm, we dramatically enhance the speed of this method.

Index Terms— clone code, PDG, isomorphism matching, MapReduce

I. INTRODUCTION

In the development of a project, maybe "copy-paste" is the largest number of operation, since it can saves developer's workload. A study about the open source software [1] found that:

Seeing the minimum copy-pasted segment size is 30 tokens, Linux gets 22.3 percent clone code in its source code, the number of FreeBSD, Apache and PostgreSQL is 20.4, 17.7, and 22.2, respectively.

Not only "copy-paste" can lead to similar code, mental macro (definitional computations frequently coded by a programmer in a regular style, such as payroll tax, queue insertion, data structure access, etc.) also brings the clone code.

The existence of similar code makes the software maintenance more difficult, when the developers want to modify the code, they may modify one place and forget somewhere else, which results the inconsistencies of the code. For a large and complex system, there are many engineers who take care of each subsystem and then modification becomes very difficult. If all the clone codes have been recorded and maintained completely, the difficulty of the modification would significantly reduce. However, in most projects, keeping all the clone code information is a laborious and costly work. In order to avoid this problem, a large amount of techniques were put forward. But the problem is precise definition of clone code. Every existing method, including line-based, token-based, AST-based and PDG-based, has its own definition of clone code. So the same source code may leads to different clone code result by different clone code method. In 2009, Roy et al [2] summarized the previous work and proposed the classification of clone code; they divided the clone code into four types:

Type-1: Identical code fragments except for variations in whitespace, layout and comments.

Type-2: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type-3: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type-4: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Generally, most methods can only detect part of clone code, according to the definition above. Line-based method can only detect Type-1 clone code, and AST-based method cannot detect Type-4 clone code.

Since each method has its own features and weaknesses, and no method is better than any other methods in every aspect [3]. So it is necessary to understand the advantages and disadvantages of each method and choose the method according to the source code and other requirements like accurateness, time complexity and space complexity.

1. This work is supported by the National Natural Science Foundation of China (Grant No. 61272415, 61272413, 61133014) and the Natural Science Foundation of Guangdong Province, China (Grant No.S2011010002708). It is also supported by the Science Program of Guangdong Province, China (Grant No.2012A080102007, 2010A011200038, 2011B090400324) and the Engineering Research Center Program of Guangdong Province, China (Grant No. GCZX-A1103).

2. Corresponding author
E-mail address: ye_lin@126.com (G. Yao).

doi:10.4304/jsw.9.6.1561-1566

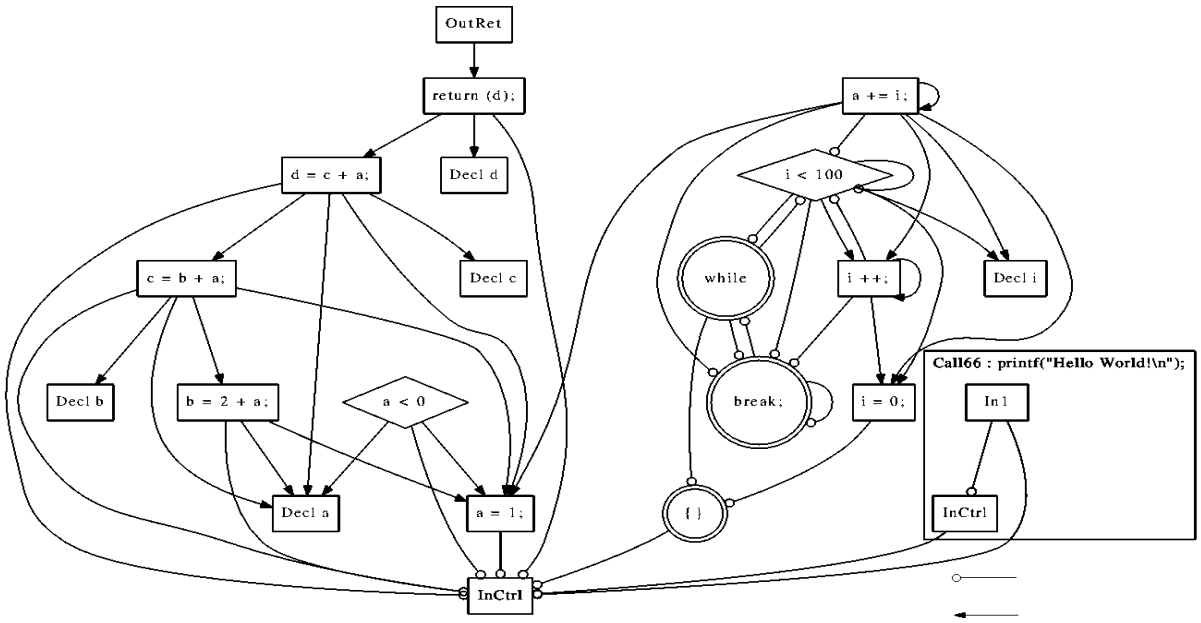

Figure.1 the PDG generated by the example of source code

The method based on PDG is able to detect semantics clone code, and more important, it can find non-contiguous clone code; while other methods are hard to find it [3].A non-contiguous clone code is a clone code departs by other codes or files, the codes from clone code are located inconsecutively. It is often produced by the modifications after pasting the clone code. Therefore, PDG-based method can detect more types of clone code.

But PDG-based method also have its own disadvantages, it's running very slowly. This approach must build the PDG from the source code, since plenty of node pairs are used as slice points, it's time consuming; and then isomorphism matching is NP-complete, it also requires high computational cost.

This paper presents an approach to parallel the isomorphism matching in the PDG. In this way, the time of isomorphism matching can be reduced. We adapt MapReduce, a prevailing parallel program paradigm, to parallel this method.

The rest of this paper is organized as follows: Section 2 introduces the background of PDG and MapReduce. Section 3 presents our algorithm. Section 4 shows the implementation which is evaluated in Section 5. Section 6 discusses related works; we conclude this paper in section 7.

II. BACKGROUND

A. Program Dependence Graph

Program dependence graph is a directed graph whose vertices present the codes in the source code, and the edges indicate the dependence between two vertices. Figure 2 is an example of source code and Figure 1 shows the PDG generated by the source code.

There are only two kinds of edges in the PDG: control dependence edge and data dependence edge. For example, given

a = b + c; s1

```
#include <stdio.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE    1024
#define DELIM   "\t"

int main(int argc, char *argv[]){
  char strLastKey[BUFFER_SIZE];
  char strLine[BUFFER_SIZE];
  int count = 0;

  *strLastKey = '\0';
  *strLine = '\0';

  while( fgets(strLine, BUFFER_SIZE - 1, stdin) ){
      char *strCurrKey = NULL;
      char *strCurrNum = NULL;

      strCurrKey  = strtok(strLine, DELIM);
      strCurrNum = strtok(NULL, DELIM); /* necessary to
check error but.... */

      if( strLastKey[0] == '\0'){
          strcpy(strLastKey, strCurrKey);
      }

      if(strcmp(strCurrKey, strLastKey)){
          printf("%s\t%d\n", strLastKey, count);
          count = atoi(strCurrNum);
      }else{
          count += atoi(strCurrNum);
      }
      strcpy(strLastKey, strCurrKey);

  }
  printf("%s\t%d\n", strLastKey, count); /* flush the count */
  return 0;
}
```

Figure.2 An example of source code

d = a * b + 1; s2

S2 has a data dependence on s1, since the value of a is used when the value of d is calculated. If s2 is executed before s1, then the value of d may be wrong. So s1 must be executed before s2. This type of dependence is data dependence.

Given

If(S) s1
d = b + c s2

S2 depends on the predicate S, if S is true, then s2 can be executed, if S is false, and then s2 will be ignored. This type of dependence is control dependence.

If the source code is changed, the PDG can only modify the corresponding part and the rest keeps the same. This feature can save plenty of time when the project updates and the PDG must be rebuilt. Therefore, PDG is widely used in the area of program optimization, code motion, vectorization, program understanding, and software engineering. A typical application of PDG is program slicing [4].

B. MapReduce

MapReduce [5] is a prevalent programming model for processing large data sets with a parallel, distributed algorithm on a cluster. It offers an ease of use programming paradigm for parallel algorithm by two user-defined functions: map and reduce. Raw data is transformed to (key, value) pairs and every map process single (key, value) pair every time.

Map: <k1, v1> → <k2, v2>

The map function is running in parallel in the cluster, and the MapReduce framework collects all pairs with the same key from the results of all the map function and passes it to a reduce function. The reduce function then generates the final result.

Reduce: <k2, v2> → <k3, v3>

Programmers write these two functions and MapReduce framework handles all the underlying work, including scheduling the parallel in the cluster and the communication between machines. In this way, the underlying architecture is transparency to the programmers, programmers can focus on what they want to do and ignore the detail of the parallel. This model also allows users to handle the partitioning and sorting keys process by customize the hashing and comparison functions, while may get a better performance than the default configuration.

Plenty of works [6] [7] [8] have been transplanted to the MapReduce, and the performance improvement convinces us to utilize this platform.

III. ALGORITHM

The granularity (e.g. function definition, begin-end block, statement sequence) must be determined before the algorithm starts. The granularity is the least unit for checking clone code; any clone code less than this size cannot be found. But as the granularity gets smaller, the running time gets longer.

First, the source code is transformed into a PDG, which is a static representation of the flow of data and control through a procedure, it's marked as s-PDG .The nodes of a PDG consist of program points constructed from the source code: declarations, simple statements, expressions, and control points. All vertexes' kind and the code they delegate are recorded for future usage.

Then pick a sub graph of the s-PDG, which correspond to a block of code in one granularity. It's marked as b-PDG.

After that, comparing s-PDG and b-PDG, to see whether there is sub graph in the s-PDG isomorphic to the b-PDG. If there is any sub graph, except b-PDG itself, in the s-PDG, isomorphic to the b-PDG, then the code corresponds to this sub graph is the clone code.

The classic algorithm of PDG-based clone code is like this, on the contrast, our algorithm cuts the s-PDG into some small graph based-on the CBCD [9], and parallels the compare of these small graphs and b-PDG. Before we state the method of cutting the s-PDG, we give the definition of the pseudo-circle used in the cutting first.

Pseudo-circle: In a graph G= (V, E), select a vertex A in V as the pseudo-center and a positive number as the pseudo-radius, then for any vertices B in V, if the Shortest Path length between A and B less than the pseudo-radius, then vertex B and the Shortest Path is in the pseudo-circle. The Shortest Path ignores the direction of the edges.

The s-PDG is cut as following:

1. Count the number of vertices with the same kind in the s-PDG.
2. Get the least vertex's kind in the s-PDG, and note it as l-kind. Then get the vertices in the b-PDG whose kind is l-kind. If there is no l-kind vertex in the b-PDG, reset the l-kind as the second least vertex's kind in the s-PDG until there are vertices with the l-kind in the b-PDG.
3. Calculate the distances between these vertices and any other vertices in the b-PDG, the maximum is set as the pseudo-radius.
4. According to the pseudo-radius above and the l-kind vertices as the pseudo-center, we can get some pseudo-circles, and these small graphs are the final result of cutting the s-PDG. We note them as the set of c-PDGs.

As in the isomorphism matching, the kind of vertices must be checked; corresponding vertex must have the same kind. So isomorphic sub graph must have an l-kind vertex, and consider the size of b-PDG, the vertices in the s-PDG too far away from the pseudo-center cannot be included in the isomorphic sub graph, so they are no longer to be considered any more.

Since the PDG of the project is divided into multiple small ones, we can parallel the process of matching the b-PDG and the set of c-PDGs. In this way, sub graph isomorphic matching, which is NP-complete, can be accomplished more efficiently.

MapReduce [5] paradigm is used to parallel this sub graph matching. Map tasks match the sub graphs, and reduce tasks gather all the matching sub graphs and output the result.

The basic flow of the algorithm is shown in the Figure.3

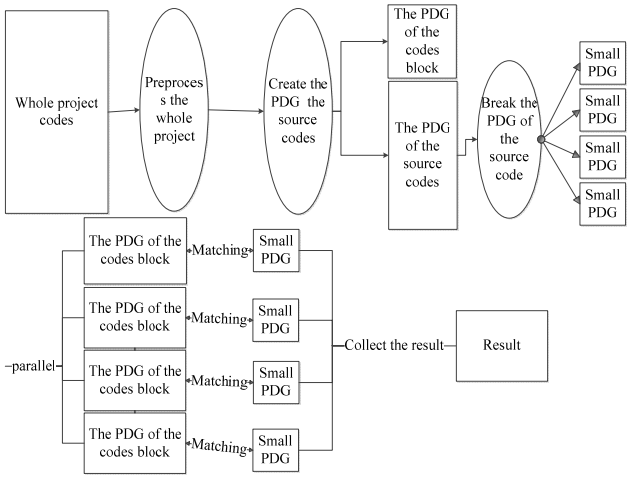

Figure.3 The basic flow of the algorithm

IV. IMPLEMENTATION

The first step is to calculate the PDG of the whole project. In our implementation, we use JavaPDG [10] to generate the PDG of the whole project. JavaPDG is a static analyzer for Java byte code, which is capable of producing various graphical representations such as the system dependence graph, procedure dependence graph, control flow graph and call graph. Java PDG can only generate PDG from Java source code, for C/C++ source code, we can use Frama-c [11] or Code Surfer [12].

JavaPDG records the category of the instructions. It can be used as the kind of the vertices, since one instruction corresponds to one vertex in the PDG. Then the PDG is cut into small graphs according to the algorithm presented in section 3. And these small graphs are given as input to the mappers. The mappers do isomorphic matching in parallel. In this way, the performance can be improved. The reducers gather all the isomorphic sub graphs and output the result.

We adapt hadoop [13], an open-source software framework supporting the MapReduce paradigm, to parallel the sub graph matching. A large amount of projects are transplanted to hadoop platform, we believe it can reduce the sub graph matching time apparently.

We use igraph [14] to matching sub graph isomorphism. Igraph is a free software package for creating and manipulating undirected and directed graphs. It includes implementations for classic graph theory problems like minimum spanning trees and network flow, and also implements algorithms for some recent network analysis methods, like community structure search.

The efficient implementation of igraph allows it to handle graphs with millions of vertices and edges. The rule of thumb is that if your graph fits into the physical memory then igraph can handle it. This feature helps igraph handle the whole PDG of a project which may have millions of vertices.

Since igraph is written in C, we use hadoop streaming, a component of hadoop which allows users to create and run jobs with any executable as the mapper and/or the reducer, to parallel the process of isomorphism checking.

V. EVALUTION

This section describes an experimental study that we conducted in order to evaluate the proposed algorithm. This experiment was performed on the virtual machines built by VMware workstation, and every virtual machine had 512MB memory. This experiment evaluated the efficiency of our algorithm by checking two open-source projects. Table.1 shows the experiment result. We evaluate the size of the project by the code lines in the project and the number of vertices and edges in the PDG of the project. We compare the time-consuming of the matching in classic PDG algorithm, and with three nodes parallel processing.

TABLE I.

CLONE DETECTION TIME

Time / Granularity (Vertices) / Source		10	50	100	200	400
SPRING-CONTEXT-3.2.1	PARALLEL	70S	78S	85S	88S	100S
	CLASSIC	625M	568M	500M	420M	334M
GITBLIT	PARALLEL	76S	82S	84S	100S	120S
	CLASSIC	630M	593M	582M	542M	523M

S presents second, M presents minute

The algorithm greatly improved the performance of the isomorphic matching; classic PDG isomorphic matching spent a couple of hours, while our parallel algorithm spent only a couple of minutes. Since we remove part of vertices in the PDG and parallel the matching process, we can get such a great improvement.

VI. RELATED WORK

We have seen plenty of improvements, such as vertices classification [9], incremental detection technique [15] and heuristic [16].

Current clone code detectors can be classified into seven kinds:

Token-based clone code detecting method [17] checks on the lexical tokens of the code, this method involves minimal code transformation.

String-based clone code detecting method [18] compares the hash code of the source code.

Abstract syntax tree based clone code detecting method[19] [20], according to the syntax tree's characteristics, this approach calculates the hash value of the code, transforms their storage forms, and then compares them node by node.

PDG based clone code detecting method [21], which is able to find semantics clone code, but the compare of sub graph is NP-complete.

Memory-state-based clone code detection [22] compares programs' abstract memory state, which is computed by a semantic-based static analyzer.

Random testing clone code detecting [23] is based on code' output values on the generated inputs.

Low-level language based clone code detecting [24] compares the low-level language code produced by complier.

All existing clone code detectors can be classified into those seven methods. Each of them has own advantages and disadvantages. No one is better than other on every aspect. What we should do is to choose the method according to the circumstance.

We also see some attempts to transplant the clone code detectors to MapReduce platform [25] [26]. We are inspired to see, with the usage of MapReduce paradigm, the methods computing the project vocabulary statistics [25] and using description logic [26] both give speed-up results.

VII. Conclusion

This paper represented and evaluated an approach to accelerate the PDG-based clone code detector. Specifically, cut the PDG into small graphs to parallel the isomorphic sub graph checking. MapReduce framework was used to parallel this checking and this makes the algorithm easily scale to get a faster speed.

The algorithm was evaluated by checking the code clones of two widely used open source projects. The result confirmed that the algorithm is effective in reducing clone code detection time. Moreover, the algorithm can accelerate by adding more machines to the cluster. Future work includes the attempt of parallel for PDG generating, which is also time consuming for a large project.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 61272415, 61272413, 61133014) and the Natural Science Foundation of Guangdong Province, China (Grant No.S2011010002708). It is also supported by the Science Program of Guangdong Province, China (Grant No.2012A080102007, 2010A011200038, 2011B090400324) and the Engineering Research Center Program of Guangdong Province, China (Grant No. GCZX-A1103).

References

[1] Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2006). CP-Miner: Finding copy-paste and related bugs in large-scale software code. Software Engineering, IEEE Transactions on, 32(3), 176-192.

[2] Roy, C. K., Cordy, J. R., & Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming, 74(7), 470-495.

[3] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and evaluation of clone detection tools. Software Engineering, IEEE Transactions on, 33(9), 577-591.

[4] Tip, F. (1995). A survey of program slicing techniques. Journal of programming languages, 3(3), 121-189.

[5] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. Communications of the ACM, 51(1), 107-113.

[6] Wang, X., Wang, Y., & Zhu, H. (2012). Energy-efficient task scheduling model based on MapReduce for cloud computing using genetic algorithm. Journal of Computers, 7(12), 2962-2970.

[7] Yang, Y., Long, X., & Jiang, B. (2013). K-Means Method for Grouping in Hybrid MapReduce Cluster. Journal of Computers, 8(10), 2648-2655.

[8] Li, R., Luo, J., Yang, D., Hu, H., & Chen, L. (2013). A Scalable XSLT Processing Framework based on MapReduce. Journal of Computers, 8(9), 2175-2181.

[9] Li, J., & Ernst, M. D. (2012, June). CBCD: Cloned buggy code detector. InSoftware Engineering (ICSE), 2012 34th International Conference on (pp. 310-320). IEEE.

[10] Shu, G., Sun, B., Henderson, T. A., & Podgurski, A. JavaPDG: A New Platform for Program Dependence Analysis.

[11] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., & Yakobowski, B. (2012). Frama-C. In Software Engineering and Formal Methods(pp. 233-247). Springer Berlin Heidelberg.

[12] http://www.grammatech.com/research/technologies/codesurfer 2013.06.

[13] http://hadoop.apache.org/ 2013.06

[14] Csardi, G., & Nepusz, T. (2006). The igraph software package for complex network research. InterJournal,Complex ystems, 1695(5).

[15] Higo, Y., Yasushi, U., Nishino, M., & Kusumoto, S. (2011, October). Incremental code clone detection: A pdg-based approach. In Reverse Engineering (WCRE), 2011 18th Working Conference on (pp. 3-12). IEEE.

[16] Higo, Y., & Kusumoto, S. (2011, March). Code clone detection on specialized pdgs with heuristics. In Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on (pp. 75-84). IEEE.

[17] Li, Z., Lu, S., Myagmar, S., & Zhou, Y. (2004, December). CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In OSDI(Vol. 4, No. 19, pp. 289-302).

[18] Baker, B. S. (1995, July). On finding duplication and near-duplication in large software systems. In Reverse Engineering, 1995., Proceedings of 2nd Working Conference on (pp. 86-95). IEEE.

[19] Wu, S., Hao, Y., Gao, X., Cui, B., & Bian, C. (2010, August). Homology detection based on abstract syntax tree combined simple semantics analysis. In Web Intelligence and Intelligent Agent Technology (WI-IAT), 2010 IEEE/WIC/ACM International Conference on (Vol. 3, pp. 410-414). IEEE.

[20] Cui, B., Li, J., Guo, T., Wang, J., & Ma, D. (2010, October). Code comparison system based on abstract syntax tree. In Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on (pp. 668-673). IEEE.

[21] Gabel, M., Jiang, L., & Su, Z. (2008, May). Scalable detection of semantic clones. In Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on (pp. 321-330). IEEE.

[22] Kim, H., Jung, Y., Kim, S., & Yi, K. (2011, May). MeCC: memory comparison-based clone detector. In Software Engineering (ICSE), 2011 33rd International Conference on (pp. 301-310). IEEE.
[23] Jiang, L., & Su, Z. (2009, July). Automatic mining of functionally equivalent code fragments via random testing. In Proceedings of the eighteenth international symposium on Software testing and analysis (pp. 81-92). ACM.
[24] Johnson, J. H. (1993, October). Identifying redundancy in source code using fingerprints. In Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1 (pp. 171-183). IBM Press.
[25] Sajnani, H., Ossher, J., & Lopes, C. (2012, June). Parallel code clone detection using MapReduce. In Program Comprehension (ICPC), 2012 IEEE 20th International Conference on (pp. 261-262). IEEE.
[26] Schugerl, P. (2011, May). Scalable clone detection using description logic. InProceedings of the 5th International Workshop on Software Clones (pp. 47-53). ACM.