2-2020

SAGA: Efficient and large-scale detection of near-miss clones with GPU acceleration

Guanhua LI
Fudan University

Yijian WU
Fudan University

Chanchal K. ROY
University of Saskatchewan

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Xin PENG
Fudan University

See next page for additional authors

Citation

LI, Guanhua; WU, Yijian; ROY, Chanchal K.; SUN, Jun; PENG, Xin; ZHAN, Nanjie; HU, Bin; and MA, Jingyi. SAGA: Efficient and large-scale detection of near-miss clones with GPU acceleration. (2020). *2020 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER): Ontario, London, February 18-21: Proceedings*. 272-283.
Available at: https://ink.library.smu.edu.sg/sis_research/5976

Author

Guanhua LI, Yijian WU, Chanchal K. ROY, Jun SUN, Xin PENG, Nanjie ZHAN, Bin HU, and Jingyi MA

SAGA: Efficient and Large-Scale Detection of Near-Miss Clones with GPU Acceleration

Guanhua Li*‡, Yijian Wu†‡, Chanchal K. Roy§, Jun Sun¶, Xin Peng†‡, Nanjie Zhan†‡, Bin Hu†‡, and Jingyi Ma†
*Software School, Fudan University, Shanghai, China
†School of Computer Science, Fudan University, Shanghai, China
‡Shanghai Key Laboratory of Data Science, Shanghai, Chin
wuyijian@fudan.edu.cn
§University of Saskatchewan, Canada
¶Singapore Management University, Singapore

***Abstract*—Clone detection on large code repository is necessary for many big code analysis tasks. The goal is to provide rich information on identical and similar code across projects. Detecting near-miss code clones on big code is challenging since it requires intensive computing and memory resources as the scale of the source code increases. In this work, we propose SAGA, an efficient suffix-array based code clone detection tool designed with sophisticated GPU optimization. SAGA not only detects Type-1 and Type-2 clones but also does so for cross-project large repositories and for the most computationally expensive Type-3 clones. Meanwhile, it also works at segment granularity, which is even more challenging. It detects code clones in 100 million lines of code within 11 minutes (with recall and precision comparable to state-of-the-art approaches), which is more than 10 times faster than state-of-the-art tools. It is the only tool that efficiently detects Type-3 near-miss clones at segment granularity in large code repository (e.g., within 11 hours on 1 billion lines of code). We conduct a preliminary case study on 85,202 GitHub Java projects with 1 billion lines of code and exhibit the distribution of clones across projects. We find about 1.23 million Type-3 clone groups, containing 28 million lines of code at arbitrary segment granularity, which are only detectable with SAGA. We believe SAGA is useful in many software engineering applications such as code provenance analysis, code completion, change impact analysis, and many more.**

***Index Terms*—clone detection, near-miss clone, segment clone, GPU acceleration, big code**

I. Introduction

Code clones, also known as duplicated code, are source code fragments that are identical or similar to each other. Code clones reflect the fact that developers intentionally copy, paste, and modify source code within or across software systems, or unintentionally write similar code according to frameworks or API specifications [1]. Code clones can be categorized according to the level of similarity [2], i.e., Type-1 are exact clones, Type-2 are parameterized clones, Type-3 are clones with further modifications (like inserting or deleting statements) based on Type-1/2, and Type-4 are clones that are not syntactically similar but semantically similar. Typically, near-miss clones are those exhibit high similarity.

Clone detection is an old engineering and research topic dated back to early 1990s [3]. Tens of clone detection approaches [4] have been proposed, supporting applications like building software libraries [5], software genealogy [6], [7], code provenance analysis [8], [9], defect analysis and predictions [10]–[12], change impact analysis [13], [14], license violation detection [15], and so on.

In the big data era, the applications of clone detection have been extended to large cross-project repositories. However, it is still challenging to detect clones in big code efficiently and effectively. It typically takes days to detect inter-project clones on 250 million lines of code (MLOC) even at the function granularity [16]. Detecting clones at a finer granularity (i.e., segment clones, which are code segments within a function but much smaller compared to the function body) is much more expensive since such detections involve much more code segments to compare and thus require enormous computation resources.

Furthermore, for Type-3 near-miss clones, segment-grained detection is particularly expensive because numerous Type-3 differences (i.e., insertion, deletion or modification of source code lines or tokens) between code segments need to be examined. While existing approaches typically rely on explicit literal boundaries (such as braces {}) to detect Type-3 clones on code blocks, we argue that detecting Type-3 clones on continuous arbitrary code segments without relying on explicit boundaries is fundamental software engineering capability that enables fine-grained clone detection, even if the final reported clones can be aligned with code block boundaries.

In this work, we propose SAGA[1], a ***S**uffix **A**rray* based clone detection tool with in-built ***G**PU **A**cceleration*, which detects Type-1/2/3 clones on arbitrary code segments. With support of GPU computation capability, it constructs a suffix array, a memory-efficient data structure, to represent the source code, and employs a GPU-accelerated algorithm to merge neighboring Type-1/2 clones for Type-3 clones. It also employs data chunking to overcome the size limit of graphics memory and supports multi-GPU to further accelerate. It has the capability of detecting Type-3 clones and detecting clones on continuous arbitrary segments.

We evaluate SAGA in terms of efficiency, scalability, recall, precision, and consistency. SAGA takes less than 11 minutes to detect clones in a 100 MLOC repository, 12 times faster

[1] https://github.com/FudanSELab/SAGA

than CloneWorks [17], 75 times faster than SourcererCC [16]. SAGA further scales up to 1 billion lines of code and finishes clone detection within 11 hours. Meanwhile, SAGA achieves comparable recall on detecting Type-1/2/3 clones, w.r.t. the state-of-the-art clone detection tools, in experiments with the BigCloneBench [18] and the Mutation and Injection Framework [19]. Manual validation of the detected clones confirms that the precision of SAGA is more than 99% at function granularity and more than 70% at segment granularity, also comparable to the competing tools. We further confirm that clone generated by SAGA is generally consistent with SourcererCC and covers those detected by CCFinder.

We also evaluate SAGA with a preliminary case study on 85,202 Java projects from GitHub, showing that SAGA facilitates various clone-based applications in large code repository which are previously impossible. We find that more than 20 million lines of code are gapped clone segments reside in long methods, which are only detectable with SAGA. Our pilot study shows that SAGA is a promising tool to support large-scale clone detection and analysis and many more software engineering applications in large code repositories.

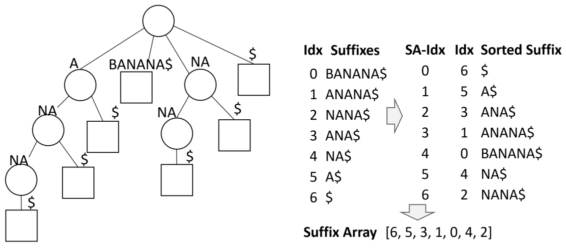
Our main contributions are the following: 1) A GPU-based clone detection tool, named SAGA, that not only detects Type-1/2/3 clones from large code repositories, but also do so at arbitrary segment granularity. This is a significant achievement over the state-of-the-art large scale Type-3 clone detectors. It is currently the only tool that detects Type-1/2 and Type-3 near-miss clones at arbitrary segment level granularity on 1 billion lines of code. 2) A comprehensive evaluation of the proposed tool with BigCloneBench and the Mutation and Injection Framework and on consistency with the competing clone detection tools. 3) A large scale preliminary case study with 1 billion lines of code across 85,202 GitHub Java projects, showing the existence and distribution of inter-project clones especially Type-3 clones at segment granularity.

The rest of the paper is organized as follows. Section II introduces preliminaries of code clone detection and GPU programming. Section III presents our approach on detecting Type-3 clone and optimizations based on GPU computation. Section IV presents relevant implementation details, including transformation rules and detection parameters. Section V shows the evaluation results with experiments on BigCloneBench and other big code repositories. Section VI reviews related work. Section VII concludes our work.

II. Preliminaries

A. Code Clone Detection

Code clones can be categorized into four types. Type-1 (a.k.a. exact) clones are code segments that are identical except for variations in white spaces and comments. Type-2 (a.k.a. parameterized) clones are code segments that are syntactically similar except for changes in identifiers, types, constants, white spaces and comments. Type-3 clones are code segments where insertions or deletions have been made based on Type-1/2 clones. Type-4 clones are code segments which are semantically similar but not syntactically or structurally similar. Near-miss clones are usually referred as those Type-3 clones that are with comparatively high syntactical similarity (e.g., 70% or more).



Idx	Suffixes	SA-Idx	Idx	Sorted Suffix
0	BANANA$	0	6	$
1	ANANA$	1	5	A$
2	NANA$	2	3	ANA$
3	ANA$	3	1	ANANA$
4	NA$	4	0	BANANA$
5	A$	5	4	NA$
6	$	6	2	NANA$

Suffix Array [6, 5, 3, 1, 0, 4, 2]

(a) Suffix Tree (b) Suffix Array

Fig. 1: An Example of Suffix Tree and Suffix Array

Clone detection approaches generally follow four steps [2], i.e., *Preprocessing* in which uninteresting files are excluded, *Transformation* in which source code is transformed to some intermediate representation, *Match Detection* in which transformed code is paired with certain data structures, and *Post-processing* in which a series of tasks are done before human-readable and meaningful clone results are generated.

Selecting the proper code representation and matching structure is essential to the performance of clone detection. For detecting clones on continuous arbitrary code segments on a large source code repository, the combination of token-based transformation and suffix-tree or suffix-array based matching is reasonable but the challenge remains on how to achieve efficiency in a large code repository and how to detect Type-3 clones.

B. Suffix Tree and Suffix Array

Suffix tree is a dictionary tree representing all the suffixes of a string, whereas suffix array is an alternative representation of suffix tree that uses the index value of the string to represent suffixes. Figure 1 shows an example of the suffix tree and the suffix array representation of the string “BANANA”.

In Figure 1(a), a path from the root to a non-leaf node represents a common substring. In Figure 1(b), a suffix array of the same string is shown. It is an array of the index value of the leading character of a suffix of the string. It ensures that, for any two values sa_i, sa_j in the suffix array sa of string *S*, if $i < j$, *suffix*(sa_i) is smaller than *suffix*(sa_j), where *suffix*(k) is the suffix string starting at index k of *S*. This means that all suffixes of the string is represented by the suffix array in the lexicographic order.

Given a string *S* and its (sorted) suffix array sa, all common substrings of *S* are the common prefixes of the suffix strings represented by “neighbor” entries of sa, e.g., *suffix*($sa[5]$) “NA” and *suffix*($sa[6]$) “NANA” share the common prefix “NA”. The length of the common prefix between *suffix*(sa_i) and *suffix*(sa_{i+1}) is defined as $height_i$, which is 2 in this case, meaning that a pair of 2-char-long common substrings “NA” exist in the string. This representation is useful in finding longest common substring, which can also be used in code clone detection. Therefore, clone detection approaches based on suffix tree or suffix array can detect any code sequences that form a clone pair no matter whether explicit code boundaries (such as "{" and "}") are specified.

Note that the suffix tree stores all suffix strings, while the suffix array stores only the index numbers of the suffix strings. Thus, suffix arrays consume much less memory than suffix trees. However, constructing a suffix array requires more computation resources. In our work, we adopt suffix array for its memory efficiency and exploit GPU parallel computation to speed up the construction process, as we show in Section III-B.

C. GPU and GPU Programming

Nowadays, general-purpose GPU, or GPGPU, are widely used for computation-intensive tasks, such as deep learning and genetic algorithm [20]. GPGPU often facilitate tens of times or even hundreds of times speedup compared to CPU computations on tasks such as floating-point operations and sorting an array.

Solutions for GPGPU programming include Nvidia's CUDA [21] and AMD's CTM [22]. CUDA enables programmers to take full advantage of both CPU and GPU in their applications and is easy to use. It provides a useful development library, which contains parallel implementations of many general-purpose functions such as array sorting, prefix-sum [23] and binary-search.

III. Our Approach

In this section, we present our approach for large scale code clone detection based on GPU acceleration.

A. Problem Analysis

Suffix-tree and suffix-array based approaches allow us to detect clone segments independent of any explicit boundaries. There are however multiple technical challenges to be addressed if we would like to apply them to large code repositories and to support detecting Type-3 clones.

First, constructing suffix trees requires significant amount of memory while constructing suffix arrays requires significant amount of computation. For instance, it was reported that suffix-tree-based approaches fail on large repositories because of huge temporary space and memory consumption [16], [24]. The memory issue motivated some attempts with suffix arrays [24] and succeeded in clone detection on the a 250MLOC code base. However, the detection took 23.5 hours (running on a 64GB RAM machine) to finish because constructing a suffix array is time-consuming. To detect clones in big code, we need an approach which is both memory and computation efficient.

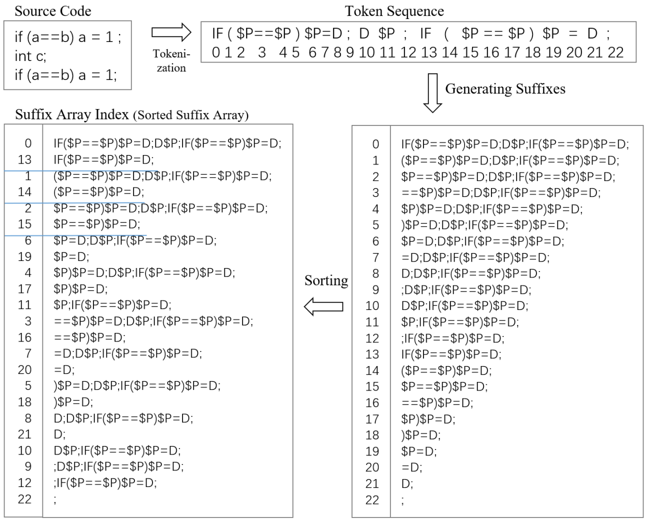


Fig. 2: Constructing a Suffix Array for Source Code

Second, it is challenging to detect Type-3 clones at segment granularity. Existing approaches detect Type-3 clones on files, functions, or code blocks but not on arbitrary code segments without explicit boundaries. Although block-grained clone reports are for a human being to read, we argue that precise fine-grained clone detection is fundamental capability for code analysis. In theory, one way to detect Type-3 clones is by merging Type-1/2 clones and calculating the code similarity [25]. However, such an approach requires extensive computation to deal with combinatorial complexity between Type-1/2 clones, which is computationally infeasible under traditional computation architecture. One way to limit the amount of Type-1/2 clones is to control the threshold on the length of minimal clone. However, it is not always feasible to set a large threshold as finer-grained clones may be missed.

Third, the suffix tree and even the suffix array may be too big to fit in the memory when we deal with large scale code. One solution is to break the data into smaller chunks so that each may fit in the main memory [25]. For instance, D-CCFinder [26] adopts a distributed, multi-computer approach and breaks the token sequence into multiple sub-sequences and processes each data chunk using a single computer. It was reported that D-CCFinder successfully detected code clones in a code base with 400 MLOC with 80 computers in about 2 days. However, such a distributed approach is expensive (by using and configuring tens of computers) and not time-efficient due to limitation on the network bandwidth.

In our approach, we design dedicated techniques to tackle the above-mentioned challenges. In the following, we present how each challenge is addressed.

B. Optimizing Suffix Array Construction

To address the first challenge, we decide to use suffix array, the memory efficient data structure, for matching and take advantage of GPU computation to speed up the construction process. The general process of constructing a suffix array is depicted with an example in Figure 2. Suffix array represent a suffix string as an *integer* of the index of each leading token in the suffix string, as marked on the left margin of the strings in Figure 2. The key part is to sort the suffixes to obtain a suffix array in which suffixes with the same prefixes become neighbors. This process typically requires a lot of computation.

To solve this problem, we adopt an existing parallelized solution, named Data Parallel Prefix-Doubling (DPPD) algorithm [27], to construct the suffix array. To the best of our knowledge, it is the first time that the algorithm is used in processing token sequences of source code. While we leave the details of the algorithm in literature, we emphasize that the algorithm is space efficient, which is essential for dealing with billions of tokens, and accelerates the construction process dramatically using the power of GPU computation. We evaluate the performance gain in a simple experiment, which shows that, with an input of 1.2 million tokens, the GPU implementation takes 2.8 seconds to construct the suffix array, more than 150 times faster than the CPU implementation which takes 439 seconds. The gain increases even more when given larger input token sequences.

C. *Detecting Type-3 Clones*

As explained in Section III-A, detecting all Type-3 clones would require dealing with the combinatorial complexity of combining all Type-1 and Type-2 clones, which is infeasible given a large code base. Thus, we design an approach which focuses on a subset of Type-3 clones which may otherwise be missed. The rational is as follows. Typically, the goal of clone detection is to report clones, either Type-1, Type-2, or Type-3, which are no shorter than a threshold on minimal clone length (l_{mc}) so that the results are not polluted by less meaningful small snippets [28]. Reporting all Type-1/2 clones is straightforward. For Type-3 clones, we partition them into two groups. One is the Type-3 clones which are constituted by at least one Type-1/2 clone which is no shorter than l_{mc}. We argue that Type-3 clones in this group are partially reported as long as qualified Type-1/2 clones are reported. The other group is the Type-3 clones which are constituted by only the Type-1/2 clones which are shorter than l_{mc}. These will be missed completely if we do not conduct Type-3 clone detection. Therefore, our goal is to systematically identify Type-3 clones in the latter group. At a high level, our approach works as follows: we detect all Type-1/2 clones to be our *candidate clones* with a threshold l_{mcc} (i.e., the minimal length for candidate clones) smaller than l_{mc}. The candidate clones that are shorter than l_{mc} are checked for merging into Type-3 clones. Figure 3 shows an example, which we explain in details in the following.

Representing candidate clones with suffix array. The source code is first transformed into a sequence of tokens (shown as ① in Figure 3) before the suffix array (shown as ②) is constructed. As explained in Section II-B, the ith value sa_i (i.e., a-f in Figure 3) are the indexes of the token sequence, meaning that the entry represents a suffix string starting at the sa_ith token, and the $height$ values (shown as ③) are the lengths of the common prefixes in the suffix strings referenced by the neighbor entries. In fact, each common prefix represents a $height$-token-long code segment. Therefore, for simplicity, we use sa_i to represent the code segment. In our example, code segment a represents a 30-token-long code segment starting at the ath token in the token sequence. Then we may recognize a pair of code segments (sa_i, sa_{i+1}) to be a candidate clone if $height_i \geq l_{mcc}$. In our example, (a, b), (c, d), and (e, f) are all candidate clones, since their height values (shown as ③) are all larger than l_{mcc} which is set to 20.

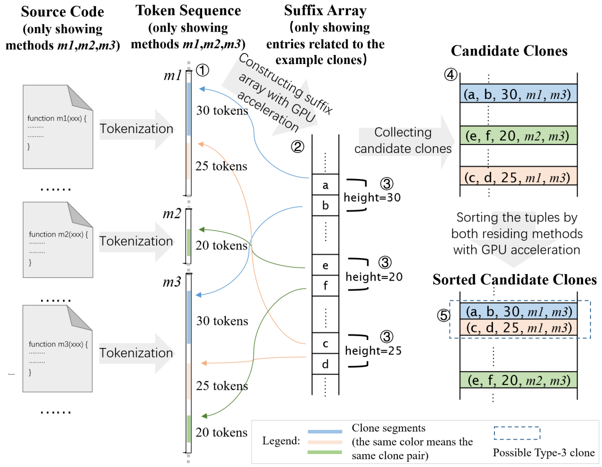


Fig. 3: Detecting Type-3 Clones: An Example

We then formally denote each candidate clone as a five-tuple $(seg_1, seg_2, len, method_1, method_2)$ (shown as ④), where seg_1 and seg_2 are code segments, len is the length of the clone, and $method_1$ and $method_2$ are the methods in which the the code segments reside. The candidate clones that are shorter than l_{mc} are then used for Type-3 clone detection, while those longer than l_{mc} are directly reported as Type-1/2 clones. Since there could be cases in which clone pairs subsume each other, the subsumed clones are removed and only the largest clone is preserved.

This approach is flexible as we can configure an appropriate value for l_{mcc} to work with defined syntactic similarity for better performance, as we discuss in Section IV-B.

Sorting candidate clones with GPU. In order to merge the candidates into Type-3 clones, we must identify candidate clones which are in the same method. This is achieved efficiently by sorting the candidate clones according to both residing methods of the two code segments. Note that sorting usually would consume a lot of time. It is however not the case, with the help of GPU (i.e., the CUDA library).

Our implementation of the sorting operation is based on a stable sort API `stable_sort_by_key`, which is very efficient even with a large amount of data (as we show in Subsection V-A). Once the candidates are sorted, we only need to consider neighboring candidates for merging into a Type-3 clone. For instance, the candidate clones $(a, b, 30, m1, m3)$ and $(c, d, 25, m1, m3)$ (shown as ⑤) are neighbors after sorting since they are in the same method, and thus may be merged. The other candidate clone $(e, f, 25, m2, m3)$ resides in a different method and thus is not a neighbor after sorting. Consequently, it will not be considered for merging with the other two example candidates.

Detecting Type-3 clones by merging candidate clones. After sorting the candidate clones, we merge the neighboring candidate clones to form Type-3 clones. The details of the merging process is shown in Algorithm 1. The input is the sorted list of candidate clones. The output is a set of Type-3 clones. We apply a similarity threshold θ for controlling the merging. Intuitively, the overall idea is to iteratively merge 'mergeable' neighbors until the next neighbor is not mergeable. Once we find a merged candidate whose $length$ is larger than l_{mc}, we add it into the result (at line 12) so that it will be reported.

Algorithm 1: Detecting Type-3 Clones

```
   Input: The sorted array $ccArray$ of the 5-tuples of all
          candidate clones
   Output: $Result$: Set of Type-3 clones
1  $Result = \varnothing$;
2  $cc$ = the first candidate clone in $ccArray$;
3  $ccToMerge = cc$; // a running cc for merge
4  while $cc \neq null$ do
5      $ccNext$ = next candidate clone in $ccArray$;
6      if IsMergeable ($ccToMerge, ccNext$) then
7          $cc$ = Merge ($cc, ccNext$);
8          $ccToMerge = ccNext$;
9      end
10     else
11         if $cc.length \geq l_{mc}$ then
12             Add $cc$ to $Result$;
13         end
14         $cc$ = next candidate clone in $ccArray$;
15         $ccToMerge = cc$;
16     end
17 end
18 return $Result$;
```

To determine whether two candidate clones are mergeable, we apply two simple tests in Algorithm 2: (1) the two clone segments in cc_1 are in the same methods as those in cc_2 (as in lines 2 to 4 for a negative condition test) and (2) the cloned code coverage is higher than θ. The coverage is determined by the total length of the two segments over the length from the start of first segment to the end of the second segment including the length of the gap in between. Therefore if the gap is too large (i.e., longer than the length from the start of the first segment to the end of the second segment multiplies $(1 - \theta)$, as shown at line 5), the clone coverage falls below θ so the clones are not mergeable.

Algorithm 2: Detect if candidate clones are mergeable

```
   Input: Two candidate clones $cc_1, cc_2$
   Output: true iff $cc_1$ and $cc_2$ are mergeable
1  Function IsMergeable ($cc_1, cc_2$)
2      if $cc_1.method_1 \neq cc_2.method_1$ or
         $cc_1.method_2 \neq cc_2.method_2$ then
3          return false
4      end
5      gap = max (gap between $cc_1.seg_1$ and $cc_2.seg_1$, gap
         between $cc_1.seg_2$ and $cc_2.seg_2$);
6      if gap > $(cc_1.length + cc_2.length +$ gap$) * (1 - \theta)$
         then
7          return false
8      end
9      return true
```

D. Optimization with Data Chunking

In existing approaches, data structures used for match detection may become too big to fit in the main memory or graphics memory as the size of source code increases. To address this challenge, we apply the idea of data chunking in the setting of GPU computation as follows. First, we divide the token sequence into data chunks so that any two chunks can be loaded in the graphics memory for suffix array construction. Then, for any two data chunks, we construct a suffix array in the graphics memory and conduct match detection. Finally, the matching results are collected to generate the final detection result.

The first step is mostly straightforward. We thus explain the next two steps in the following. Assume that we have N data chunks of equal size. Let c_i and c_j $(1 \leq i, j \leq N, i \neq j)$ be two arbitrary chunks. We fit c_i and c_j in the graphics memory for detecting clones across these two chunks and the clones within either c_i or c_j. In general, we need to perform the above for $N * (N - 1)/2$ pairs of data chunks to get all clones detected. A similar approach is used in D-CCFinder [26].

We conduct multiple experiments in order to determine the right size of each data chunk. Our experiments reveal that constructing a suffix array for 200 million tokens at one time is safe and stable and achieves high utilization of one NVidia card with 6GB graphics memory. Based on this observation, we decided to divide the token sequence in chunks of 100 million tokens each and designed a scheduler to feed two data chunks to one graphic card at a time.

Although the chunking approach addresses the above-mentioned challenges adequately, we further apply the following two optimizations to exploit the capability of the GPU computation architecture. (1) *Multi-GPU support*. Since each data chunk can be processed independently, they can be executed by multiple GPU concurrently. Therefore it is feasible to leverage multi-GPU support. (2) *Pipeline solution in CPU-GPU architecture*. It is even more effective to perform pipeline-like execution on both CPU and GPU, i.e., in a producer/consumer pattern. The CPU is responsible for preprocessing and transformation, whereas GPU is responsible for suffix array construction and match detection. When the GPU is constructing suffix array for a tokenized data chunk, the CPU immediately schedules the next data chunk for tokenization. This pipeline solution makes full use of the CPU-GPU architecture and further improves the performance.

IV. Implementation

In this section, we present relevant implementation details.

A. Tokenization

Our tokenization module has a lightweight parser which does not rely on any third-party parsers. The parser employs predefined tokenization rules as stated in Table I. The rules share similar ideas with those used in CCFinder [25]. However, our rules are simpler and do not require language-dependent pretty-printing beforehand. The tokenization module is generally language-independent except for a small number of

TABLE I: Tokenization Rules

Operation	Examples/Description
R1 Accessibility or modifier keywords removal	Remove keywords like: `protected`, `static`, `final`.
R2 Keyword replacement	Some keywords, such as `try`, `catch` and `switch`, are replaced by their own tokens.
R3 Basic data types unification	Basic data types, such as `int`, `long`, `Int`, `Long`, are replaced by a specific token.
R4 Identifier & data replacement	All class names, variable names, and constants are replaced by a specific token.
R5 Block boundary recognition	Method boundaries are transformed to a specific token. Boundaries (e.g., braces in Java) for a block less than 4 lines are removed.

customized keywords and method boundary tokens. Methods' boundaries are used for for locating a clone segment and for finding neighboring clones in candidate merging.

Compared to other preprocessing tools such as TXL [29] used in other clone detection tools [16], [25], [30], our lightweight parser runs more efficiently because 1) it does not parse complex code structures and 2) it does not pretty-print the source code but conduct tokenization directly with a list of language keywords and symbols.

B. Deciding the Minimal Length of Candidate Clones

The minimal length of candidate clones l_{mcc} determines the smallest code segment to be considered. Setting a smaller l_{mcc} helps to detect Type-3 clones since smaller fragments of code may be detected and can participate in the merging process. However, an overly small l_{mcc} may produce too many tiny candidate clones that are unlikely meaningful, which harms the execution time and the precision.

Since it is commonly accepted that a clone needs to be at least 50 tokens long to be meaningful [31], we assume that a Type-3 clone must also have at least 50 cloned tokens, apart from at least one gap. Assume the similarity threshold is 0.7. Then the typical Type-3 clone should be 71 tokens long $(50/0.7)$, including a 21-token gap. Candidate clone should not be too small and should be comparable to a typical gap. According to the results of the threshold experiment, we set l_{mcc} to 20, taking recall rate and efficiency into consideration.

V. Evaluation

We evaluate SAGA on its execution time, recall, and precision with BigCloneBench, the Mutation and Injection Framework, and an open-source code base consisting of 1 billion lines of code, and also analyze the consistency with the competing tools. We further conduct a preliminary case study on 85,202 projects with SAGA to show its usefulness in big code analysis.

A. Execution Time and Scalability

As SAGA targets large code repositories, we evaluate the execution time on source code datasets of various scales.

Datasets: One source code dataset is the IJaDataset [32], which contains about 250 million lines of Java source code mined from SourceForge and Google Code. The full IJaDataset and its subsets are often used for evaluating execution time and scalability of clone detection tools [16], [30], [33]. Another source code dataset we use is 85,202 non-fork Java projects downloaded from GitHub, containing about 1 billion lines of source code. Among them, there are about 8,000 projects (300MLOC) that have earned more than 100 stars each. Combining the 300MLOC with the IJaDataset, we have 550MLOC, which is of medium size between 250MLOC and 1 billion LOC for scalability tests and also used in recall experiments later.

Environment and Settings: We run SourcererCC[2], CCFinderX[3], NICAD[4], Deckard[5], CloneWorks[6], and our own tool SAGA, on the above-mentioned source code datasets. All experiments are run on a single workstation equipped with an Intel i7 CPU (8-core, 16-thread), 32GB RAM, 1TB SSD, and one NVIDIA GTX 1080Ti graphics card with 6GB graphics memory. We configure SourcererCC, CCFinder, and CloneWorks to use all 16 CPU threads in order to fully utilize computation capability of the workstation.

We also compare execution time with iClone [34], CCAligner [30], and Oreo [33]. iClone is a tool originally designed for incrementally detecting evolution of clones within a project although it is also able to detect clones on single-version projects. Considering its limited scalability, we did not run the tool on our own workstation but took the execution time reported in the literature [16]. CCAligner and Oreo are tools optimized for detecting large gap clones and weakly Type-3 clones, respectively. Since we do not have these tools at hand, we estimate the execution time of these two tools according to the literature, with comparison to our own execution time of SourcererCC.

SAGA was configured to detect clones longer than 50 tokens, with minimal candidate clone length 20 tokens. Similarity was set to 0.6, a comparatively low threshold that may produce more outputs. So the execution time should be close to the upper bound. If similarity were set higher for better precision, execution time would be lower. Settings for SAGA and the competing tools are listed in Table II.

Results: The execution time results are shown in Table III. SAGA finishes clone detection on 100 MLOC source code in 10m57s, about 50-70 times faster than SourcererCC and CCFinder [7], and about 12 times faster than CloneWorks[8]. At the 100MLOC scale, NICAD, Deckard, and iClones fail to parse the code or report out-of-memory errors.

[2] SourcererCC 2.0, https://github.com/Mondego/SourcererCC/releases

[3] CCFinderX 10.2.7.4, http://www.ccfinder.net/ccfinderxos.html

[4] NICAD 4.0, https://www.txl.ca/nicaddownload.html

[5] Deckard 2.0, https://github.com/skyhover/Deckard/releases

[6] CloneWorks 0.3, https://jeffsvajlenko.weebly.com/cloneworks.html

[7] It was previously reported [16] that CCFinder took twice as much time as SourcererCC. But in our setting, CCFinder outperforms SourcererCC on the 100 MLOC dataset. It is reasonable because we provide more main memory so that CCFinder needs less temporary disk space.

[8] In our replication experiment, CloneWorks works faster in Conservative(C) configuration than in Aggressive(A) configuration, which is different from previous reported results [17]. It is possibly because we provide more memory and more powerful CPU which significantly accelerate the "build" phase.

TABLE II: Tool Settings for Scalability Test

Tool	Settings
SAGA	Min clone length 50 tokens, min candidate length 20 tokens, min similarity 0.6, function granularity
SourcererCC	Min length 6 lines, min similarity 0.7, function granularity
CloneWorks	Min length 6 lines, min clone length 50 tokens, min similarity 0.7, function granularity
CCFinder	Min clone length 50 tokens
NICAD	Min clone length 6 lines, identifier abstraction none, similarity 0.7
Deckard	Min clone length 50 tokens, similarity 0.85, 2 token stride

We investigated the distribution of time in each step of SAGA. It spent 3m48s (35% of total time) in preprocessing and tokenization, 1m2s (9%) in constructing suffix array, 5m0s in finding Type-1/2 clones (46%), and 1m7s (10%) in generating Type-3 clones and post-processing. The two phases utilizing GPU take less than 19% of total time while CPU takes the rest.

Furthermore, SAGA finishes clone detection on the IJaDataset (with 250 MLOC) in 1 hour 24 minutes. According to the literature [16], [17], SourcererCC and CloneWorks are the only ones that scales up to this size with a single workstation[9]. CloneWorks takes 18h27m in conservative(C) mode and 8h3m in aggressive(A) mode. However, SourcererCC terminates with a parser error when we try to run it with the IJaDataset. Since it was reported [16] that SourcererCC finished in 4 days 12 hours on the 250 MLOC dataset and 1 day 12 hours 54 minutes on the 100 MLOC dataset, we assume SourcererCC executes 2.9 times longer on the 250 MLOC dataset than on the 100 MLOC dataset. Therefore, considering SourcererCC actually finished in 12h27m on 100MLOC dataset in our setting, we estimate SourcererCC should spend no shorter than 36 hours (12h27m * 2.9 = 36.1h) on the IJaDataset.

As for CCFinder, although it was reported that CCFinder failed on IJaDataset due to insufficient disk space [16], we still had a try since CCFinder outperformed SourcererCC in our environment on the 100 MLOC dataset. Unfortunately CCFinder ends up with an internal error on 250MLOC. Again, we estimate that the execution time of CCFinder on 250MLOC would be no less than 25 hours according to the fact that CCFinder's execution time was 0.7 times of SourcererCC's when previously dealing with the 100 MLOC dataset.

We then run SAGA on the 550 MLOC dataset and it finishes clone detection in 4 hours 50 minutes after processing 2.08 billion tokens. We further run SAGA on the dataset with 1 billion lines of code except that the detection is at segment granularity so that the results can be used for our preliminary case study. SAGA successfully finishes clone detection within 11 hours.

Alternative Settings and Results: We also tried different settings for SAGA on the 550 MLOC dataset. For detecting Type-1/2 clones only, SAGA finished in 4 hours 18 minutes. This confirms that generating Type-3 clones takes around 10% of total detection time in SAGA. For detecting clones at segment granularity, SAGA took 4 hours 36 minutes to finish, under the same settings except for the similarity threshold set to 0.7. This is about 15 minutes faster than detection at function granularity because we have optimized the merging-based Type-3 clone detection at segment granularity with GPU whereas the detection of Type-3 clones at function granularity is based on calculation of code lines coverage which is purely CPU-based and we are not able to further optimize.

Furthermore, changing l_{mc} from 50 to 40 while keeping $l_{mcc} = 20$ slightly increases the execution time (i.e., only several minutes longer averagely). However, when l_{mcc} is reduced from 20 to 15, we soon run out of disk space due to enormous increase of candidate clones. This indicates that the number of candidate clones is essential for the efficiency.

B. Recall

In order to evaluate the recall of our approach, we use two benchmarks, i.e., BigCloneBench [18], a well-recognized code clone benchmark with real clones, and the Mutation and Injection Framework [19], a synthetic benchmark that evaluates a tool's recall with automatically-generated artificial clones. These benchmarks are widely used in previous work [16], [30], [31], [33]. Since the benchmarks only provide validated clones at function granularity, we set our tool to report clones at function granularity by checking whether the Type-1/2 clone coverage (by tokens) in a method is above predefined similarity threshold.

For BigCloneBench, we consider benchmarked clones with at least 6 pretty-printed lines and 50 tokens. This setup has been accepted as a standard when measuring recall [16], [31], [33]. We also replicated the recall experiments for any tools that we have at hand. We report in Table IV the recall of SAGA with various configurations, along with the recalls of other tools. All results are categorized by clone types, i.e., Type-1, Type-2, Very Strong Type-3, Strong Type-3, Moderately Type-3, and Weakly Type-3/Type-4 [16], [18].

It shows that the recall of SAGA is comparable to the competing tools when sim=0.6 and l_{mc}=40. When the similarity threshold changes from 0.6 to 0.7, the recall on Type-1/2 clones almost does not drop but there is an obvious drop of recall on Type-3 clones. The reason of the drop is that SAGA computes the similarity differently from how the benchmark does, especially in the Strong Type-3 (ST3) part which is at the edge of the 0.7 threshold. We have set l_{mcc}=20 so SAGA ignores code clone candidates shorter than 20 tokens. This continuous-token-similarity strategy is a stronger constraint than the overall-similarity in the bags-of-tokens or statement-level approaches. For example, 0.7 similarity in 50 tokens need only 35 tokens to be the same regardless whether the 35 tokens are continuous or not. If the 35 tokens are continuous, the code segment will look more similar. In SAGA's strategy, the 0.6 similarity on a 50-token-long function requires a 30-token-long continuous token sequence, which is still a strong similarity constraint.

[9] Gabel et al. [35] extended Deckard to deal with big code. However we currently are not able to have access to the tool. Nevertheless, our tool is potentially a substitution for Deckard in their code clone studies.

TABLE III: Execution Time

LOC	SAGA	SourcererCC	CCFinder	NICAD	Deckard	iClones	CCAligner	Oreo	CloneWorks(C)	CloneWorks(A)
1K	1s	3s	2s	1s	1s	1s*	1s**	–	3s	3s
10K	1s	5s	5s	1s	4s	1s*	1s**	–	3s	3s
100K	4s	7s	10s	5s	32s	2s*	3s**	–	4s	4s
1M	9s	37s	39s	12s	27m12s	MEM*	>20s**	–	26s	26s
10M	59s	12m21s	6m30s	19m49s	ERROR	–	>5m**	–	11m12s	11m3s
100M	10m57s	12h27m	9h49m	ERROR	–	–	–	–	2h19m	1h55m
250M	1h24m	36h**	>25h**	–	–	–	–	>222h**	18h27m	8h3m
550M	4h50m	–	–	–	–	–	–	–	–	–

* Data reported in the literature ** Data estimated based on reported time with consideration of environment differences

When the l_{mc} changes from 40 to 50, the drop of recall is caused by different calculation of number of tokens. SAGA has its own code parser and calculates total length of the function differently. SAGA only tokenizes the body of a method but the competing tools and BigCloneBench itself consider both the body and the signature of the method. Therefore, the number of tokens of SAGA is slightly smaller than those of the tools and the benchmark, causing SAGA to miss some target code segments with the length slightly longer than 50 tokens. Typically, SAGA would get around 40 tokens for a 50-token method. Therefore, we decide that setting l_{mc}=40 is fair and reasonable.

On overall, the recall of SAGA is comparable to that of the competing tools on not only Type-1/2 clones but also Type-3 clones. CloneWorks and NICAD are top in recalling VST3 and ST clones. SAGA performs well on VST3 clones, following CloneWorks and NICAD, and performs similar to SourcererCC on ST3 clones. If SAGA is further tuned in the calculation of the number of tokens to match the benchmark criteria, then setting l_{mc}=50 can perform better in the benchmark test.

Also note that SAGA achieves the same recall of the benchmarked clones embedded in the 550 MLOC dataset (which we used earlier in the scalability experiment), showing that the recall of our tool is stable as the dataset grows.

For the Mutation and Injection Framework, we run SAGA with 1,050 mutants generated from 7 functions and 150 mutators each, including Type-1/2 clones and Type-3 clones with at least 0.7 similarity. The recall is perfect or near perfect on all three types, which is the comparable to SourcererCC, NICAD, and CCAligner as previously reported [16], [30], and better than CCFinder which recalls 0 Type-3 mutants.

C. Precision

The precision is measured manually by validating a random and statistically significant sample [16], [30], [33] of the clones detected by SAGA. Each clone is examined independently by two experts. If there is a conflict, a final decision is made after discussion with a third expert. The principle rule for judging is based on the overall similarity between the two clone segments and on whether they perform similar tasks. Since SAGA can be configured to output clones at either function granularity or segment granularity, we evaluate the precision separately.

Function Granularity: Five of the authors divided in two groups spent 3 weeks in inspecting 12,700 clones from the detection results of SAGA in the recall experiment with BigCloneBench. The result shows that SAGA has a high precision of 99%. False positives are mainly due to very short consecutive assignment statements, different API calls repeated in the `case` clause in `switch-case` structure, or extremely long single statements doing a lot of string concatenating. We also sampled 400 clones, also a statistically significant sample as accepted in the literature [30], [33], from the detection results detected by SourcererCC and NICAD in the recall experiment with BigCloneBench. We find that the precision is lower than previously reported, so we adopt previously reported precisions in Table IV for comparison. Precision results of CloneWorks, CCAligner, and Oreo are also taken from the literature. The results show that SAGA performs comparably with the other tools.

Segment Granularity: We run SAGA and CCFinder on real-world projects downloaded from GitHub. Since the whole dataset is too big for CCFinder, we randomly select 2.3 MLOC as input to CCFinder. We randomly check 400 clones reported by each tool and find that the precision of CCFinder is lower than previously reported [16]. So we adopt reported precision in Table IV. The results suggest that SAGA has comparable precision with CCFinder. Note that CCFinder only outputs Type-1/2 clones, whereas SAGA outputs not only Type-1/2 but also Type-3, which is more challenging in terms of precision. The other tools do not support segment-grained detection, so the precisions for segment granularity are left empty.

D. Consistency across Tools

We compare the real clones reported by SAGA, SourcererCC, and CCFinder, in order to check whether the results are consistent.

We choose SourcererCC and CCFinder for comparison because SourcererCC is a large-scale Type-1/2/3 clone detector that can work at function granularity and CCFinder detects clones at segment granularity. Our tool covers the detection capability of both tools.

Since intra-project clones are relatively easier to detect, we focus on inter-project clones in the GitHub dataset. Since the full dataset (1 billion lines of code) is too big for SourcererCC and CCFinder to finish clone detection timely, we randomly choose 353 Java projects (about 10 MLOC) to be our experiment dataset. However, CCFinder ends up with an internal error on this dataset possibly due to lack of robustness. Therefore, we run CCFinder multiple times and each time on

TABLE IV: Recall and Precision on BigCloneBench

Tool	Threshold/Setting	T1(35,800)		T2 (4,574)		VST3 (4,156)		ST3 (15,013)		MT3(79,922)		WT3/T4(7.75M)		Precision	
		%	#	%	#	%	#	%	#	%	#	%	#	Func	Sgmt
SAGA	sim=0.6, l_{mc}=40, function	100	35,781	100	4,552	95	3,932	60	9,030	10	7,686	0	5,936	99	–
SAGA	sim=0.7, l_{mc}=40, function	100	35,781	98	4,504	91	3,769	45	6,715	5	4,141	0	2,171	99	–
SAGA	sim=0.6, l_{mc}=50, function	99	35,552	96	4,375	93	3,866	56	8,396	8	6,055	0	3,126	99	–
SAGA	sim=0.7, l_{mc}=50, segment	–	–	–	–	–	–	–	–	–	–	–	–	–	75
SourcererCC*	sim=0.7, MIT=1	100	35,797	98	4,462	93	3,871	61	9,099	5	4,187	0	2,005	98	–
SourcererCC$^+$	sim=0.7, MIL=6	98	35,185	85	3,849	89	3,680	58	8,651	7	5,211	1	80,520	83	–
CCFinder*	MIT=50	100	–	93	–	62	–	15	–	1	–	0	–	–	72
CloneWorks(A)*	sim=0.7, MIT=1	100	35,777	99	4,544	98	4,090	93	13,976	3	2,700	0	35	99	–
NICAD*	sim=0.7, MIL=6	100	35,769	99	4,541	98	4,091	93	13,910	1	671	0	12	99	–
CCAligner*	sim=0.7, $q = 6$, $e = 1$	100	–	99	–	97	–	70	–	10	–	–	–	80	–
Oreo*	sim=0.55, MIT=15, τ=0.6	100	35,798	99	4,547	100	4,139	89	13,391	30	23,834	1	57,273	90	–

* Data reported in the literature (same configuration) + Data collected in the authors' replication experiment

one part of the projects, resulting in much less inter-project clones.

Since it is infeasible to manually cross-check all clones detected by the three tools across 353 projects, we select 5 target projects, with stars from 95 to 4,900, to manually check the relevant clones (i.e., the clone pairs that touch at least one of the 5 selected projects).

In SourcererCC's results, there are 6,749 clone pairs relevant to the 5 projects; there are 9,548 in SAGA's results. CCFinder only detects parts of the code separately and therefore the number of detected clones is much smaller. Although we do not consider CCFinder to be comparable to SourcererCC and SAGA, we randomly check 400 clone pairs in CCFinder's results and find that all are reported by SAGA, showing a good one-way consistency.

As for SourcererCC, we check 3,000 true positive clones that are at least 10 lines long reported by SourcererCC and find that 2,641(88%) of them are reported by SAGA as segment clones. The missing clones are mainly of two categories. The first are those small methods whose lengths in tokens are below the minimal clone threshold (l_{mc}). The other are Type-3 clones that consist of multiple very small separated identical code snippets that fall below the minimal candidate clone threshold (l_{mcc}). Reversely, we check 1,000 true positive clones reported in SAGA and find that, if the segment clone reported by SAGA covers almost the whole method, it is also reported by SourcererCC; if the segment clone is in a long method (e.g., $\geq$ 50 lines) and only covering a small part of the residing methods, it is not reported by SourcererCC. This difference shows the finer-grained detection capability of SAGA.

In general, SAGA shows good consistency with SourcererCC at function granularity and also detects clones at segment granularity.

E. A Preliminary Case Study

We further explore how SAGA is useful in large real-world code base and thus conduct a preliminary case study on the 85,202 GitHub Java projects to exemplify the use of the tool. We remark that due to limited scalability of existing code clone detection tools, previous studies are limited to either clones in a small code base (less than 6,000 Java projects) [36] or clones at file granularity [37], [38]. It is thus not clear whether the results obtained with a small code base is valid in the large or how finer-grained code clones exist in the large. Using SAGA, we aim to analyze how pervasive code clones at segment granularity are in big code (with one billion lines-of-code) and see how these clones distribute within and across the projects.

We run SAGA with $l_{mc} = 50$, $l_{mcc} = 20$, and $similarity = 0.7$ and obtain more than 142 million clone groups containing about 327 million clone instances (i.e., clone code segments). 98% clone groups contain 4 or less clone instances, among which 79.4% contain only 2 clone instances. About 0.3% (i.e., more than 220 thousand) clone groups contain more than 20 clone instances each. Furthermore, about 93.4% of the clone groups are inter-project while only 6.6% contain clones all from the same project. We find this result reasonable because a clone group is regarded as inter-project as long as it contains code segments from 2 or more projects, even if the majority of the clone instances in the clone group come from the same project. 79.3% clone groups contain clone code from exactly 2 projects. Nearly 99.5% of the clone groups contain clone code from 4 or fewer projects, showing that most clones only spread in a small range of projects.

Some code snippets even show up thousands of times in hundreds of projects. For example, a code snippet appears 9,901 times across 2,538 projects, which we recognize as part of the `hashCode` function that can be automatically generated by Eclipse. Some clones are accidental [1], [36] since we normalize the name of variables and method calls into tokens (which is a common practice [25]). Tokenization can be customized [28] to reduce reports of uninteresting accidental clones. Some frequently-occurring clones are API usage patterns, common algorithms, and similar switch/case structures. We find that some interesting clones, such as repeated functionalities, are not so frequently occurring and are not widely spread. They often appear dozens of times across several projects. This partly supports that clones may be related to specific business domains [36]. Initial investigation suggests that many of them are worthy of consideration for better code reuse or maintenance.

Each of the clone groups contains either Type-1/2 clones or Type-3 clones (but not both) since we detect Type-3 clones by merging Type-1/2 clones. There are 1.23 million Type-3 clone groups, containing 28 million lines of code. It is worthwhile to note that 1.04 million Type-3 clone groups (i.e., 84% among all Type-3 clone groups) contain at least one clone instance that covers less than 70% lines of code in its residing function. These are clones that hide in long functions, which are only detectable with SAGA. Type-3 clones at segment granularity exhibit fine-grained similar source code and reveal clues of the 'copy-paste-modify' coding practice, such as modifications of loop conditions, inserting logging statements, or changing an intermediate local value to the return of a function (by adding a `return` keyword). SAGA opens a door for further investigation of source code replications within and across projects at the segment granularity in big code repositories.

We remark that SAGA enables clone detection in big code and is useful for many software engineering applications such as code provenance analysis, license violations, mining the seeds of new APIs, code completion, and many more. We will address them in future work.

F. Threats to Validity

The performance of clone detection tools is affected by the configurations and the environments. To achieve a fair comparison to other competing tools, both the benchmark dataset and the tools are carefully tuned and, whenever possible, experiments reported in the literatures are replicated in order to minimize the threats to internal validity.

Another threat lies in the precision measurement. Although we stick to explicit criteria for judging clones and try to minimize human errors by employing three individual judgers, we still find it difficult to confidently determine whether a reported clone is true or false, especially when Type-3 clones at segment granularity are considered. We find that the decision of a 'true clone' is sometimes related to the purpose of the engineering task. So we save the judgement as an open problem for future work.

We have not evaluated SAGA in programming languages other than JAVA, which is a threat to external validity. However, our tool is extensible to other languages and we have already implemented a C/C++ version in an industrial collaboration.

VI. Related Work

Many code clone detection approaches and tools have been proposed, including text-based [28], [39], token-based [11], [25], [34], tree-based [40]–[43], and graph-based ones [44], [45]. Matching techniques in clone detection include substring comparison [39], longest common subsequence (LCS) [28], subtree comparison [41], [42], hashing [46], [47], location sensitive hashing (LSH) [48], suffix tree [25], [43], suffix array [24], [49], indexing [46], etc.

Research work has aimed at detecting code clones at large code repositories (e.g., 100-400 MLOC) on a single work station [16], [17], [25] or with a network of computers [26], [50]. Déjàvu [38] scales up to 4.5 million *projects* and shows a map of code clones in the projects but it works only at file granularity with the support of SourcererCC. A number of empirical studies [1], [5], [36], [51]–[53] have been conducted on various large scale of real-world code repositories. Commercial products such as BlackDuck [54] and FOSSID [55], integrate even larger code repositories with more than tens of millions of projects to provide code scan services but do not support detecting Type-3 clones at segment granularity.

Previous research also suggested the importance of detecting Type-3 clones. NICAD [28] accurately detects near-miss intentional clones in a line-based approach. CCAligner [30] is suitable for detecting large gapped Type-3 clones. Oreo [33] aims at the twilight zone of weakly Type-3 clones, combining techniques of machine learning, information retrieval, and software metrics. But these tools do not scale to big code. SourcererCC [16] scales up to 400 MLOC but need to work with specified code segment boundaries.

Deep learning-based clone detection techniques [56]–[62] arise in recent years. These tools have an advantage over traditional tools in detecting weakly Type-3 or Type-4 clones by extracting higher-dimensional information from the code. These techniques are usually supervised, which need specifying code boundaries and labeling them to make training set. The detection result is limited to train set and not able to reach to arbitrary fragment granularity. They make use of GPU computation as a tool for a learning-based tasks, such as training. Seldom do we find approaches that directly leverage GPU computation power for code transformation or matching tasks. The only work we find is Lavoie's [63], which reports a GPU implementation of dynamic programming matching algorithm for clone detection. However, we are unable to find reports on its precision, recall, or scalability in the literature.

VII. Conclusion and Future Work

Tackling the difficulty in detecting Type-3 clones at segment granularity on big code, we propose SAGA, a suffix array based code clone detection tool that leverages GPU computation power to accelerate the detection process. The tool scales up to 1 billion lines of code within 11 hours' execution time. The recall and precision are comparable to the state-of-the-art clone detection tools. It is the only tool to-date that can deal with large data of that magnitude and at the same time detect near-miss clones at arbitrary segment level of granularity. Thanks to SAGA's scalability and capability, we conduct a preliminary case study on inter-project clones with 85,202 Java projects, showing its usefulness for many software engineering applications which enlightens our future work.

Acknowledgments

This work was supported by National Key R&D Program of China under Grant No. 2017YFB1002000 and Shanghai Science and Technology Development Funds (18DZ1112100 and 18DZ1112102).

References

[1] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," in *2005 International Symposium on Empirical Software Engineering (ISESE 2005), 17-18 November 2005, Noosa Heads, Australia*, 2005, pp. 376–385.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *SCHOOL OF COMPUTING TR 2007-541, QUEEN"s UNIVERSITY*, vol. 115, 2007.

[3] B. S. Baker, "A program for identifying duplicated code," in *Computer Science and Statistics: Proc. Symp. on the Interface*, March 1992, pp. 49–57.

[4] D. Rattan, R. K. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information & Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[5] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries - an empirical study on 13, 000 projects," in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 387–391.

[6] R. K. Saha, M. Asaduzzaman, M. F. Zibran, C. K. Roy, and K. A. Schneider, "Evaluating code clone genealogies at release level: An empirical study," in *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010, Timisoara, Romania, 12-13 September 2010*, 2010, pp. 87–96.

[7] M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 187–196.

[8] D. Steidl, B. Hummel, and E. Jürgens, "Incremental origin analysis of source code files," in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, 2014, pp. 42–51.

[9] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Trans. Software Eng.*, vol. 31, no. 2, pp. 166–181, 2005.

[10] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 422–432.

[11] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[12] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug-proneness and late propagation tendency of code clones: A comparative study on different clone types," *Journal of Systems and Software*, vol. 144, pp. 41–59, 2018.

[13] M. Mondal, C. K. Roy, and K. A. Schneider, "An insight into the dispersion of changes in cloned and non-cloned code: A genealogy based empirical study," *Sci. Comput. Program.*, vol. 95, pp. 445–468, 2014.

[14] N. Bettenburg, W. Shang, W. M. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at the release level," *Sci. Comput. Program.*, vol. 77, no. 6, pp. 760–776, 2012.

[15] D. M. Germán, M. D. Penta, Y. Guéhéneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*, 2009, pp. 81–90.

[16] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 1157–1168.

[17] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with cloneworks," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, 2017, pp. 27–30.

[18] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 476–480.

[19] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*, 2009, pp. 157–166.

[20] P. Krömer, J. Platos, and V. Snásel, "Genetic algorithm for clustering accelerated by the CUDA platform," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, SMC 2012, Seoul, Korea (South), October 14-17, 2012*, pp. 1005–1010.

[21] Nvidia, "Nvidia cuda compute unified device architecture, programming guide version 1.0," http://www.nvidia.com, 2007.

[22] AMD/ATI, "Ati ctm guide technical reference manual, version 1.01," http://ati.amd.com, 2006.

[23] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley, August 2007, ch. 39, pp. 851–876.

[24] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *16th European Conference on Software Maintenance and Reengineering, CSMR 2012, Szeged, Hungary, March 27-30, 2012*, 2012, pp. 309–318.

[25] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[26] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pp. 106–115.

[27] W. Sun, "Using GPU to accelerate suffix array construction," in *7th International Conference on Biomedical Engineering and Informatics, BMEI 2014, Dalian, China, October 14-16, 2014*, 2014, pp. 677–682.

[28] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, 2008, pp. 172–181.

[29] J. Cordy, "The txl programming language," http://www.txl.ca/.

[30] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "Ccaligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 1066–1077.

[31] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. Software Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[32] A. S. E. Group, "Ijadataset 2.0," http://secold.org/projects/seclone, January 2006.

[33] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: detection of clones in the twilight zone," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 354–365.

[34] N. Göde and R. Koschke, "Incremental clone detection," in *13th European Conference on Software Maintenance and Reengineering, CSMR 2009, Architecture-Centric Maintenance of Large-SCale Software Systems, Kaiserslautern, Germany, 24-27 March 2009*, 2009, pp. 219–228.

[35] M. Gabel, J. Yang, Y. Yu, M. Goldszmidt, and Z. Su, "Scalable and systematic detection of buggy inconsistencies in source code," in *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, 2010, pp. 175–190.

[36] M. Gharehyazie, B. Ray, and V. Filkov, "Some from here, some from there: cross-project code reuse in github," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 291–301.

[37] J. Ossher, H. Sajnani, and C. V. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, 2011, pp. 283–292.

[38] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "Déjàvu: a map of code duplicates on github," *PACMPL*, vol. 1, no. OOPSLA, pp. 84:1–84:28, 2017.

[39] S. Ducasse, M. Rieger, and S. Demeyer, “A language independent approach for detecting duplicated code,” in *1999 International Conference on Software Maintenance, ICSM 1999, Oxford, England, UK, August 30 - September 3, 1999*, 1999, pp. 109–118.

[40] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, 1998, pp. 368–377.

[41] B. Biegel and S. Diehl, “JCCD: a flexible and extensible API for implementing custom code clone detectors,” in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, 2010, pp. 167–168.

[42] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, “DECKARD: scalable and accurate tree-based detection of code clones,” in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 96–105.

[43] R. Koschke, R. Falke, and P. Frenzel, “Clone detection using abstract syntax suffix trees,” in *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pp. 253–262.

[44] J. Krinke, “Identifying similar code with program dependence graphs,” in *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE’01, Stuttgart, Germany, October 2-5, 2001*, 2001, pp. 301–309.

[45] Y. Higo and S. Kusumoto, “Code clone detection on specialized pdgs with heuristics,” in *15th European Conference on Software Maintenance and Reengineering, CSMR 2011, 1-4 March 2011, Oldenburg, Germany*, 2011, pp. 75–84.

[46] B. Hummel, E. Jürgens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, 2010, pp. 1–9.

[47] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou, “Inferring repository file structure modifications using nearest-neighbor clone detection,” in *19th Working Conference on Reverse Engineering, WCRE 2012, Kingston, ON, Canada, October 15-18, 2012*, 2012, pp. 325–334.

[48] M. S. Uddin, C. K. Roy, K. A. Schneider, and A. Hindle, “On the effectiveness of simhash for detecting near-miss clones in large scale software systems,” in *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, 2011, pp. 13–22.

[49] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida, “SHINOBI: A tool for automatic code clone detection in the IDE,” in *16th Working Conference on Reverse Engineering, WCRE 2009, 13-16 October 2009, Lille, France*, pp. 313–314.

[50] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, “DCCD: an efficient and scalable distributed code clone detection technique for big code,” in *The 30th International Conference on Software Engineering and Knowledge Engineering, Hotel Pullman, Redwood City, California, USA, July 1-3, 2018.*, 2018, pp. 354–353.

[51] M. Mondal, C. K. Roy, and K. A. Schneider, “Does cloned code increase maintenance effort?” in *11th IEEE International Workshop on Software Clones, IWSC 2017, Klagenfurt, Austria, February 21, 2017*, 2017, pp. 38–44.

[52] C. K. Roy and J. R. Cordy, “An empirical study of function clones in open source software,” in *WCRE 2008, Proceedings of the 15th Working Conference on Reverse Engineering, Antwerp, Belgium, October 15-18, 2008*, 2008, pp. 81–90.

[53] C. K. Roy and J. R. Cordy, “Near-miss function clones in open source software: an empirical study,” *Journal of Software Maintenance*, vol. 7, no. 3, pp. 165–189, 2010.

[54] S. S. I. Group, “Blackduck,” https://www.blackducksoftware.com, 2017.

[55] FOSSID, “Fossid,” https://fossid.com, 2016.

[56] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 933–944.

[57] D. Perez and S. Chiba, “Cross-language clone detection by learning over abstract syntax trees,” in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada.*, 2019, pp. 518–528.

[58] L. Büch and A. Andrzejak, “Learning-based recursive aggregation of abstract syntax trees for code clone detection,” in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, 2019, pp. 95–104.

[59] J. Zeng, K. Ben, X. Li, and X. Zhang, “Fast code clone detection based on weighted recursive autoencoders,” *IEEE Access*, vol. 7, pp. 125 062–125 078, 2019.

[60] L. Li, H. Feng, W. Zhuang, N. Meng, and B. G. Ryder, “Cclearner: A deep learning-based clone detection approach,” in *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 249–260.

[61] G. Zhao and J. Huang, “Deepsim: deep learning code functional similarity,” in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 141–151.

[62] H. Wei and M. Li, “Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code,” in *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 2017, pp. 3034–3040.

[63] T. Lavoie, M. Eilers-Smith, and E. Merlo, “Challenging cloning related problems with gpu-based algorithms,” in *Proceeding of the 4th ICSE International Workshop on Software Clones, IWSC 2010, Cape Town, South Africa, May 2010*, 2010, pp. 25–32.