On Accelerating Ultra-Large-Scale Mining

Ganesha Upadhyaya
Iowa State University
ganeshau@iastate.edu

Hridesh Rajan
Iowa State University
hridesh@iastate.edu

***Abstract*—Ultra-large-scale mining has been shown to be useful for a number of software engineering tasks e.g. mining specifications, defect prediction. We propose a new research direction for accelerating ultra-large-scale mining that goes beyond parallelization. Our key idea is to analyze the interaction pattern between the mining task and the artifact to cluster artifacts such that running the mining task on one candidate artifact from each cluster is sufficient to produce results for other artifacts in the same cluster. Our artifact clustering criteria go beyond syntactic, semantic, and functional similarities to mining-task-specific similarity, where the interaction pattern between the mining task and the artifact is used for clustering. Our preliminary evaluation demonstrates that our technique significantly reduces the overall mining time.**

***Keywords*-Big Code, acceleration, clustering, Boa, analysis**

I. Introduction

Mining open source software repositories such as GitHub is valuable and can be leveraged to help software engineering tasks, e.g. for defect prediction [1], bug fix suggestions [2], specification inference [3], etc. The approaches that leverage these open source repositories perform mining and analysis that cuts across projects.

The current infrastructure support for ultra-large-scale mining leverages parallelization techniques such as map-reduce [4]–[6]. We propose a new direction for accelerating ultra-large-scale mining based on artifact similarities. Our idea is to use artifact similarities to cluster artifacts such that it is sufficient to perform mining on one candidate in each cluster and extrapolate the mining results to others. In this way, ultra-large-scale mining can be accelerated.

Software source code is an important artifact that is mined often. There exists many techniques to identify code clones [7]. Syntactic clones identify codes that look alike. Semantic clones identify semantically similar codes (control or data flow similarities). Functional clones [8] detect codes that are functionally similar by comparing inputs and outputs of methods. Although syntactic and semantic clones may result in the same output for a mining task and hence they can be used to cluster and accelerate the ultra-large-scale mining, they may be less beneficial. This is because in case of syntactic clones, the amount of acceleration is limited by the amount of copy-and-paste code and in case of semantic clones, only a subset of analysis can accelerate, for instance control flow only analyses. Functional clones may not result in the same output for a mining task, hence they cannot be used to cluster. This has led us to go beyond syntactic, semantic, and functional clones to develop a notion of *mining-task-specific similarity*.

Given a mining task, a set of artifacts can be considered similar, if the mining task has similar interactions with the artifacts. When a mining task is run on an artifact, one can observe the interaction between the mining task and the artifact. The interaction is captured by the execution trace, where the trace describes, what parts of the artifact are of interest to the mining task, and whether such parts exists in the artifact. The interaction (or the execution trace) can be captured by running the mining task on the artifact, however the challenge is to determine the interaction without running the mining task on the artifacts.

Our observation is that, by analyzing the mining task, it is possible to extract the parts of the artifacts that are of interest to the mining task. By utilizing this knowledge, it is possible to capture the interaction by performing a light-weight traversal of the artifact. This traversal identifies the parts of the artifact that are of interest to the mining task. Upon identifying these parts, the parts that are not of interest to the mining task can be removed, resulting in a reduced artifact that only contains parts relevant for the mining task. This reduced artifact is used to represent the interaction between the mining task and the artifact.

Given a mining task that is required to be run on thousands of artifacts, the artifacts with similar interactions are clustered together, such that the mining task is required to be run on only one candidate from each cluster to produce the mining result and the results for other candidates in the same cluster can be produced using extrapolation.

This paper makes the following contributions:

- We propose a new research direction for accelerating ultra-large-scale mining by *task-specific clustering*.
- We introduce the notion of *interaction pattern graph* that represents the interaction between the mining task and the artifact, however we leave the details of soundly extracting the graph a subject of the future work.
- We present two case-studies that demonstrates the usage of our technique.

II. Approach

An overview of our approach is shown in Figure 1. Given a mining task and a large collection of artifact graphs[1], a light-weight traversal is performed on each artifact graph that identifies the parts relevant for the mining task and removes

[1]An artifact graph represents the relation between parts in the artifact. For instance, in case of source code artifact, a control flow graph represents the control flow relation between program statements.

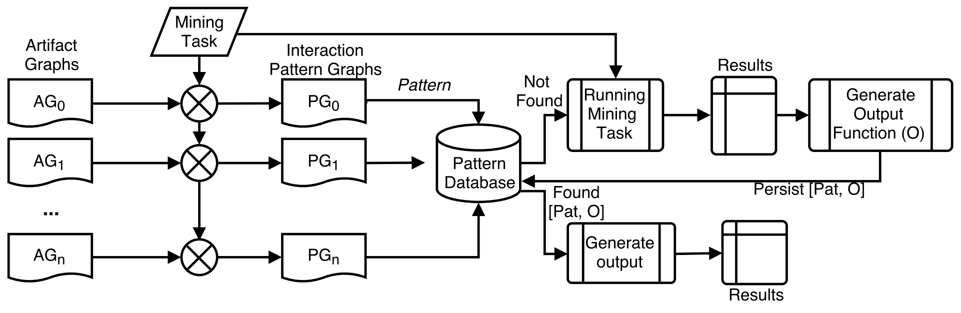

Fig. 1. Overview of the approach: accelerating ultra-large scale mining using the interaction pattern between the mining task and the artifacts.

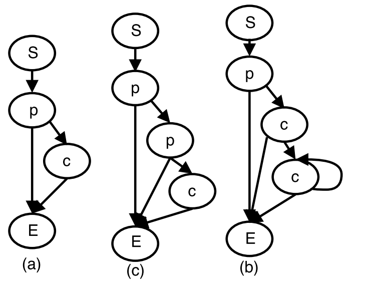

Fig. 2. Top three interaction pattern graphs for the API Precondition Mining Task. Here, S and E are start and end nodes, p is the node with predicate expression, and c is the node that calls an API method.

the irrelevant parts to produce an *interaction pattern graph*[2]. Upon generating the interaction pattern graph, we check if the pattern graph is already seen before while mining other artifact graphs, if not then we run the mining task on the original artifact graph to generate the output. An output function that provides expressions to generate the output is constructed. A simple output function is the mining task itself (as we see later in the realization of this model, we use the mining task as the output function). We persist the pattern along with its output function, such that next time when a match happens, we extract and apply the output function to generate the result, instead of running the mining task.

```
 1 public void body(String namespace, String name, String text)
 2   throws Exception {
 3   String namespaceuri = null;
 4   String localpart = text;
 5   int colon = text.indexOf(':');
 6   if (colon >= 0) {
 7     String prefix = text.substring(0,colon);
 8     namespaceuri = digester.findNamespaceURI(prefix);
 9     localpart = text.substring(colon+1);
10   }
11   ContextHandler contextHandler = (ContextHandler)digester.peek();
12   contextHandler.addSoapHeaders(localpart,namespaceuri);
13 }
```

Fig. 3. Code snippet from Apache Tomcat GitHub project.

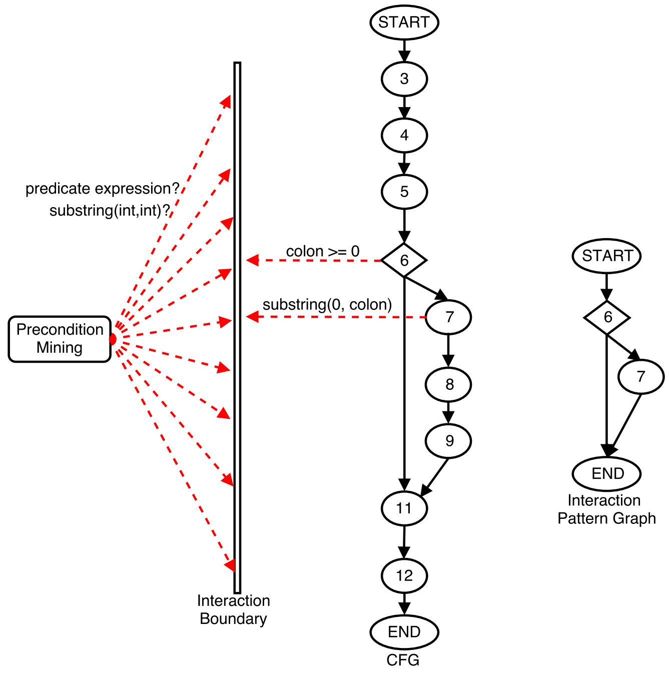

Fig. 4. The interaction of the precondition mining task with the CFG of the code shown in Listing 3.

A. *Interaction Pattern Graph*

An interaction pattern graph is a reduced artifact graph that retains only those nodes of the artifact graph that are relevant for the given mining task. To illustrate, consider a task of mining API preconditions. API preconditions are the conditions that must be satisfied before calling an API method. API preconditions can be inferred by looking at the guard conditions at the API method call sites [3]. Input to the mining task is a set of API methods whose preconditions are to be mined and a large number of client projects that calls the API methods. The technique builds the control flow graphs (CFG) of the client methods and performs a dominator analysis to determine all control dependent nodes of the API method call node in the CFG, that contains predicate expressions. The output of the mining task is a set of predicate expressions on which the API method call is control dependent (in a way these predicate expressions guard the API method call).

Here, the artifacts are the client methods and the mining task queries method statements that calls API methods and conditional statements that guard the API method calls. To understand the interaction between the mining task and the artifact, consider the example code shown in Figure 3. This example calls $substring(int, int)$ API method from $java.lang.String$ at line 7. This API method call is guarded by the precondition at line 6, whose predicate expression is $colon >= 0$. Now, let us consider Figure 4 that shows the interaction between the *Precondition Mining* task and the CFG of the method

[2]An interaction pattern graph represents the interaction between the mining task and the artifact graph.

shown in Figure 3. When the *Precondition Mining* task is run on the CFG, it queries for predicate expressions and $substring(int, int)$ API method calls. Each node in the CFG responds to the queries if they have predicate expressions or $substring(int, int)$ API method calls. For the CFG shown in the figure, only node 6 and node 7 responds successfully as node 6 contains predicate expression $colon >= 0$ and node 7 makes the $substring(int, int)$ API method call. The interaction can be summarized using the interaction pattern graph shown in Figure 4. The interaction pattern graph contains only those nodes that successfully responded to the mining task queries (START and END are two special nodes).

B. Generating The Interaction Pattern Graph

We briefly present the process of generating an interaction pattern graph. For generating the interaction pattern graph, we perform a static analysis of the mining task to extract a set of rules that helps to identify nodes in the artifact graph that are relevant for the mining task. We then perform a light-weight traversal of the artifact graph, where at each node we execute the rules, mark the nodes as relevant/irrelevant, and remove the irrelevant nodes to produce an interaction pattern graph.

1) Extracting Rules To Infer Interaction Pattern Graphs: Mining a software artifact requires traversing the artifact graph and querying the nodes to extract some information. For instance, in our *API Precondition Mining* task, the artifact graph is the CFG of the method. The listing below shows the pseudo code of the traversal in the mining task.

```
API_precondition_mining (ArtifactGraph G) {
  output: {predicate expressions, API method calls}
  For each node in G
    if (node is a predicate expression)
      add predicate expression to output
    else if (node is an API method call)
      add API method call to output
  return output
}
```

Our claim is that it is possible to extract a set of rules from the traversals. For instance, the pseudo code above has two rules: *Rule 0: node is a predicate expression* and *Rule 1: node is an API method call*. We deduce these rules as follows. We analyze the body of the traversal and extract program paths. These program paths have path conditions that must be satisfied for that path to be taken. Our idea is to use the conjunction of path conditions as rules to identify the relevant nodes. The above pseudocode has two paths: one that takes the *if* branch and other path that takes the *else* branch. The path condition for the first path is *(node is predicate)* $\bigwedge$ ¬*(node is an API method call)*. The path condition for the second path is ¬*(node is predicate)* $\bigwedge$ *(node is an API method call)*. In this way, by enumerating the paths in the traversal body and collecting the path conditions, one can construct a set of rules that helps to infer the interaction pattern.

2) Generating Interaction Pattern Graph: Upon extracting the rules from the traversals, we perform a light-weight traversal of the CFG to generate the interaction pattern graph. In this traversal we keep only those nodes for which the rules evaluates to *true*. For instance, consider the CFG shown in Figure 4. Only *node 6* and *node 7* are kept in the interaction pattern graph, because for only these two nodes the rules evaluates to *true*.

3) Model Realization: In our realization of the model shown in Figure 1, we have used the mining task itself as the output generating function, where upon generating the pattern graph, we run the mining task on the pattern graph to produce output. Note that, running the mining task on the interaction pattern graph has lower complexity than running it on the original artifact graph, because the interaction pattern graph contains fewer nodes that are of interest to the mining task.

III. Preliminary Results

In this section, we present two case studies that illustrate the usage of our technique.

TABLE I
CASE STUDY RESULTS SUMMARIZED. IPA IS OUR APPROACH (INTERACTION PATTERN ANALYSIS), G IS GAIN, R IS REDUCTION.

	Analysis Time (s)			Graph Size		
	Baseline	IPA	%G	Baseline	IPA	%R
CS1	1555.579	309.731	80	52,475,484	17,945,817	66
CS2	235.524	131.160	45	52,475,484	21,631,192	59

A. Case Study 1. Mining API Preconditions

In this case study we use *Mining API Preconditions*, described in §II-A, that mines API preconditions of a given API method using the client methods that call the API method. Here, the API preconditions are the predicate expressions that guard the API method calls in the client methods. This mining task traverses the CFG of each client method, identifies nodes that have API method calls and collect the predicate expressions on which the API method call node is control dependent using a dominator analysis. The mining task outputs the normalized predicate expressions as preconditions for a given API method call.

We have used Boa [4] for writing the mining query and we run the mining query on the Boa [4] SourceForge dataset that contains over 7 million client methods. We compared our approach with a baseline that runs the precondition mining task on the client methods sequentially. The baseline took *1555.579 seconds* to collect *5965* preconditions at *2240* client methods of $substring(int, int)$ API method. On the same dataset, our approach also collected the same *5965* preconditions at *2240* client methods as baseline and the mining task took *309.731 seconds*. In this case, our technique is able to reduce the mining time by 80%. On further investigation we found that the total number of CFG nodes in the CFGs were *52,475,484* and the total number of nodes in the interaction pattern graphs of those CFGs were *17,945,817*, that is, over 66% reduction in terms of graph size. To summarize, our approach accelerates the precondition mining task by running the mining task on the interaction pattern graph that contained only API method call nodes and predicate expression nodes.

Figure 2 shows top three interaction pattern graphs that appeared in the client methods that calls the $substring(int, int)$

API method. We argue that candidates in each cluster shows similar behaviors with respect to the mining task. Studying the candidates in each cluster may itself be a new research direction for exploring and answering mining task related questions. For instance, we found that the interaction pattern graph shown in Figure 2(a), almost all the time provides predicate expressions that are generic to the API method and not specific to the client method that calls the API method.

B. Case Study 2. Vulnerability Detection Using Taint Analysis

In this case study our mining task detects possible vulnerabilities using a light-weight taint analysis. If the source of the value of a varible x is untrustworthy, we say that x is tainted. For instance, line $x = System.out.readLine()$ tells that x is tainted. Taint Analysis attempts to identify the variables that have been "*tainted*" with user controllable inputs and traces them in the program. If a tainted variable gets passed to a public sink without first being sanitized, it could cause a vulnerability. A public sink could be an output buffer or a console. For instance, in line $System.out.println(x)$, tainted x is passed to the console. Given a method, this mining task outputs the number of possible vulnerabilities in that method by performing an intra-procedural taint analysis.

We wrote the mining task in Boa and we ran it on the Boa's SourceForge dataset, which contains over 7 million methods. The baseline that runs the mining task on methods sequentially took *235.524 sec.* and our approach took *131.160 sec.* We were able to match all the *14309* possible vulnerabilities identified by both the approaches. The speed up that our technique achieved is close to 45%. On further investigation, we found that baseline CFGs contained *52,475,484* nodes and the corresponding interaction pattern graphs contained *21,631,192* nodes, that is, 59% reduction in terms of graph size. We verified several of these *14309* possible vulnerabilities, however it is impossible to verify them all.

In summary, our two case studies suggest that our technique of clustering based on the interaction between the mining task and the artifact can significantly accelerate the ultra-large scale mining. However, the acceleration that can be obtained depends both on the reduction in the graph size from artifact graph to interaction pattern graph, and complexity of the analysis.

IV. Related Works

Accelerating Software Analysis. Kulkarni *et al.* [9] accelerates program analysis in Datalog by running the analysis offline on a corpus of training programs to learn analysis facts over shared code and then reuses the learnt facts to accelerate the analysis of other programs that share code with the training corpus. When compared to their approach, our approach does not require programs or artifacts to share code or other artifacts. Reusing analysis results to accelerate interprocedural analysis by computing partial or complete procedure summaries [10] is also studied. However, to best to our knowledge there is no technique that can benefit analysis across programs and cluster programs specific to analysis.

Clustering or Detecting Similar Code. Similar code or code clones includes "*look alike*" codes that are textually, syntactically, structurally similar and codes that are behaviorally or functionally similar. Existing approaches of identifying code clones can be categorized based on the types of intermediate representations they use [7]: token-based, AST-based, and graph-based. There are also other approaches that goes beyond structural similarity: code fingerprints [11], behavioral clones [8], [12], and run-time behavioral similarity [13]. Clone detection techniques are agnostic to the mining task that is performed on the artifacts. Our technique groups artifacts that produces similar result for the given mining task. Artifacts produce similar results, if they show similar interaction patterns for the given mining task.

V. Conclusion and Future work

This emerging result shows that task specific clustering can cluster behaviorally similar artifacts for the given mining task, and can help to accelerate ultra-large-scale mining. We have shown usefulness of task-specific clustering for two representative tasks, but more work is needed to understand the characteristics of the mining tasks that can benefit from task-specific clustering. Other avenues of the future work includes: i) producing an efficient output function for extrapolating the output, ii) efficiently persisting the interaction pattern graph and the output function, and iii) efficient graph comparison.

VI. Acknowledgments

This research has been supported in part by the US National Science Foundation (NSF) under grants CCF-15-18897, CNS-15-13263, and CCF-14-23370.

References

[1] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *ESE'11*.

[2] Benjamin Livshits and Thomas Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE'13*.

[3] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining Preconditions of APIs in Large-scale Code Corpus. In *FSE'14*.

[4] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE'13*.

[5] Sushil Bajracharya, Joel Ossher, and Cristina Lopes. Sourcerer: An Infrastructure for Large-scale Collection and Analysis of Open-source Code. *Sci. Comput. Program. 2014*.

[6] Georgios Gousios. The GHTorrent dataset and tool suite. In *MSR '13*.

[7] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comput. Program. 2009*.

[8] Rochelle Elva and Gary T. Leavens. Semantic Clone Detection Using Method IOE-behavior. In *IWSC'12*.

[9] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating Program Analyses by Cross-program Training. In *OOPSLA'16*.

[10] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*.

[11] Collin McMillan, Mark Grechanik, and Denys Poshyvanyk. Detecting Similar Software Applications. In *ICSE'12*.

[12] F. H. Su, J. Bell, and G. Kaiser. Challenges in Behavioral Code Clone Detection. In *SANER'16*.

[13] John Demme and Simha Sethumadhavan. Approximate Graph Clustering for Program Characterization. *TACO'12*.