Parallel and Distributed Code Clone Detection using Sequential Pattern Mining

Ali El-Matarawy
Faculty of Computers and Information, Cairo University

Mohammad El-Ramly
Faculty of Computers and Information, Cairo University

Reem Bahgat
Faculty of Computers and Information, Cairo University

ABSTRACT
This research presents a parallel and distributed data mining approach to code clone detection. It aims to prove the value and importance of deploying parallel and distributed computing for real-time large scale code clone detection. It is implemented this approach in a family of clone detectors, called PD EgyCD (Parallel and Distributed Egypt Clone Detector). In this approach, This research builds on an earlier work of the authors for code clone and plagiarism detection using sequential pattern mining by adding parallelism and distribution to our earlier tool EgyCD. Our approach uses data mining through a tailored Apriori-based algorithm for code clone detection. And it uses parallelization and distribution to achieve excellent performance to scale up to clone detection on very large systems. This approach has been implemented as a database application which leverages the capabilities of modern database tools. Two versions have been developed of this distributed technique. The first one uses client-server technique in which all clients and the server deal with only one database. The second one uses agents where each client acts as a separate agent and has its own database and after working on a sub-problem, it submits its partial solution to the server to finally get the complete solution (set of code clones). Experiments show that agents technique is faster than client-server one. Distribution enhances performance very much. Speed improvement is a function of the number of clients/agents used. Our conclusion is that data mining, combined with parallel and distributed computing, can efficiently be deployed for code clone detection of very large systems.

Keywords
Code clones, textual approach, lexical approach, syntactic approach, clone types, parallel code clone detector, distributed code clone detector, clone relation terminologies, data mining, apriori property, sequential pattern mining.

1. INTRODUCTION
It is very common in computer programming to copy a part of the program from one place and paste it in another place and then adapt it to fit in the new place. This happens for a variety of reasons [1]. Hence, software often includes multiple copies of the same piece of code, which are known as code clones. Research suggests that between 7% and 23% of the code-base of typical software system consists of code clones [2, 3]. Sometimes code clones are created for legitimate reasons, but other times they are not and they deteriorate the quality of the code. One of the main drawbacks of code clones is that the developer should modify multiple copies of the same piece of code if a change or a correction is needed in one copy. Often this deteriorates code quality if some clones were forgotten and left unchanged [1].

A recent study on industrial systems shows that inconsistent changes/updates to cloned code are frequent and lead to severe unexpected behavior [4]. Other studies suggest that code clones can make maintainability difficult [5, 6] and introduce subtle errors in software systems [7, 8]. Thus code clones are considered one of the bad "smells" of code [9]. Hence, detection, monitoring, removal and management of code clones has become an important topic in software maintenance and evolution research that received significant attention in the last decade in particular [9].

Several techniques have been proposed to find code clones and their locations in the code [1].

In a previous research, we developed EgyCD [17], a sequential clone detection tool that employs a tailored Aprori-based sequential pattern mining (SPM) algorithm for code clone detection. The tool was implemented as a database application that can detect 100% of Type 1, Type 2 and Type 3 code clones in multiple languages provided that a language definition table is properly filled with language specific information. EgyCD demonstrated the applicability and benefits of using data mining techniques for code clone detection.

However, due to the exhaustive nature of Apriori-based algorithms, speed and scalability were issues that needed improvement in EgyCD. The same problem faces many other code clone detection techniques. For such tools to operate on very large scale code bases and provide real-time response, employing parallel and distributed techniques is crucial. However, not all algorithms and source code representations can easily be distributed. We have the advantage that Apriori-based algorithms, and the associated representation of source code as items and itemsets, can easily be parallelized and distributed.

This research introduces two improvements that enhance code clone detection using data mining in EgyCD. The first enhancement is using parallel programming with multiple threads on a single machine. This version of PD EgyCD increased speed by almost 20% as experiments show. The second enhancement is using distribution across multiple machines. There are two versions of this enhancement. The first is a client-server version and the second uses agents. In the agents version, agents work autonomously independent of each other to develop separate parts of the solution. Finally, they send their partial solutions to a server to integrate them and produce a complete solution. Distributed versions scale up seamlessly with the growth of available hardware.

An experiment has been performed to test and compare these versions using 2151 randomly selected files from Java JDK version 6, containing about 310861 lines of code. The size of these files is 21.8 MB. All EgyCD versions (stand alone, multi threaded and distributed) have been tested. Results are very promising. They suggest that despite the exhaustive nature and slow performance of Apriori-based algorithms in clone detection compared to other techniques, distribution can

scale up these techniques to detect clones in very large projects.

The rest of this paper is organized as follows: Section 2 presents code clones basic definitions and terminology. Section 3 briefly discusses related work. Section 4 is an overview for data mining techniques relevant to code clone detection. Section 5 introduces our approach for code clone detection using multi threading and distribution. Three case studies are reported in Section 7. Section 8 presents an analysis of the results and discusses advantages and limitations of our approach. Finally Section 9 is the conclusion and future work.

2. BASIC DEFINITIONS

To make this paper self-contained, we start by introducing the basic established terminology used in the field of code clone detection and management. We mainly followed the same basic definitions of Roy et. al. [1] and Roy and Cordy [3].

Definition 1: Code Fragment. A code fragment is a continuous part of the source code, maybe consists of one line or more. It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements.

Definition 2: Code Clone. A Clone occurs when a code fragment is identical to another code fragment according to some basic criteria, this criteria may be syntactically and/or semantically identical, or a little bit changing in renaming identifiers,…., etc.

Definition 3: Clone Types. There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a code fragment and pasting into another location. In the following we provide the types of clones based on both the textual (Types I to III) [10] and functional (Type IV) similarities.

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

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

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

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

Clone Relation Terminologies.
Clone detection tools report clones in the form of Clone Pairs (CP) or Clone Classes (CC) or both. These two terms speak about the similarity relation between two or more cloned fragments. The similarity relation between the cloned fragments is an equivalence relation (i.e., a reflexive, transitive, and symmetric relation) [11]. A clone-relation holds between two code portions if (and only if) they are the same sequences. Sequences are sometimes original character strings, strings without whitespace, sequences of token type, transformed token sequences and so on. In the following we define clone pair and clone class in terms of the clone relation [12]:

Clone Pair: A pair of code portions/fragments is called a clone pair if there exists a clone-relation between them, i.e., a clone pair is a pair of code portions/fragments which are identical or similar to each other.

Clone Class: A clone class is the maximal set of code portions/fragments in which any two of the code portions/fragments hold a clone-relation, i.e., form a clone pair.

3. RELATED WORK

Various approaches to code clone analysis have been proposed including string–based, token-based, syntax tree-based, metrics-based, graph-based (e.g., program dependence graphs; PDGs [13-15]) and hybrid approaches. Most of these were developed in sequential tools. This limits the scalability and performance of such tools and their applicability to very large code bases.

Very few approaches and tools utilized parallel and distributed programming. This is due to the inherited complexity in such programming approaches on one hand and the difficulty of breaking and distributing the used source code representation across multiple machines (e.g., syntax tees, PDGs, etc.) on the other hand. Application of such techniques on top of existing clone detection tools will enable very large scale clone detection across multiple projects and huge code bases and real-time and online duplicate code detection.

Livieri *et. al.* developed an approach for distributed large scale code clone detection. It is an extension to a successful token-based code clone detection tool, CCFinder. They implemented a distributed version of the tool named, D-CCFinder. They applied this tool successfully to a vast collection of open source programs. [18]

Hummel *et. al.* implemented an incremental index-based distributed code clone detection approach to detect Type I and Type II clones. They applied their approach on a case study of 73 MLOC of Eclipse, using 100 machines with detection time of 36 minutes. Since the clone index can be updated incrementally while the software changes, cloning information can be kept accurate at all times. Distribution is supported by the fact that clone index can be distributed across different machines, enabling index creation, maintenance and clone retrieval to be parallelized [16].

Our approach similarly seeks scalability and performance improvement via parallelization and distribution. But it is different in that it is the first that combines parallelization and distribution with data mining for code clone detection.

4. DATA MINING OVERVIEW.

Data mining [20, 21] is the process of extracting interesting (non-trivial, implicit, previously unknown and potentially useful) information or patterns from large information repositories such as: relational database, data warehouses, XML repository, etc. Also data mining is known as one of the core processes of Knowledge Discovery in Database (KDD).

Sequential Pattern Mining.

Definition 1: Sequential pattern mining [21] is trying to find the relationships between occurrences of sequential events, to find if there exists any specific order of the occurrences.

In data mining [22] frequent itemsets are used to illustrate relationships within large amounts of data. The classical example is the analysis of the buying-behavior of customers.

The database consists of a set of transactions, and each transaction is a set of items from a universal itemset *I*.

The goal is to find itemsets I that are subsets of many transactions T in the database D, (I ∈ T). An itemset is called frequent, if it occurs in a percentage that exceeds a certain given support count σ [19]:

$$\sigma(I) = \frac{|\{T \in D\}|\{I \subseteq T\}|}{|D|} \geq \sigma$$

EgyCD is not interested in the percentage of itemsets. Instead it is interested in their count

$$\sigma(I) = |\{T \in D\}|\{I \subseteq T\}| \geq \sigma \text{ where } \sigma > 1$$

Most SPM algorithms are based on Apriori algorithm [21]. Sequential pattern mining was first introduced in [14] by Agrawal. Given the transaction database with three attributes customer-id, transaction-time and purchased-items, the mining process were decomposed into five phases:

Sort Phase: the original transaction database is sorted with customer-id as the major key and transaction time as the minor key, the result is set of customer sequences.

L-itemsets Phase: the sorted database is scanned to obtain large 1-itemsets according to the predefined support threshold..

Transformation Phase: the customer sequences are replaced by those large itemsets they contain, all the large itemsets are mapped into a series of integers to make the mining more efficient. At the end of this phase the original database is transformed into set of customer sequences represented by those large itemsets.

Sequence Phase: all frequent sequential patterns are generated from the trans-formed sequential database.

Maximal Phase: those sequential patterns that are contained in other super sequential patterns are pruned in this phase, since it is only interested in maximum sequential patterns.

Since most of the phases are straightforward, researches focused on the sequence phase in [14].

5. GENERAL DESCRIPTION of EGYCD.

Here, a describing of the original sequential EgyCD algorithm is presented first before describing PD EgyCD versions in the next section. It is exactly the same as we presented it first in [17]. For source code, each statement is a transaction. An itemset is a sequence of statements. Each statement is treated as a line of code (LOC). Following Apriori-based approaches, EgyCD builds up larger itemsets (clones in this case) from combining smaller ones and then efficiently searches the source code to verify their presence. If present, they are then used to form larger clones or itemsets. EgyCD tool consists of four steps:

a. The user selects the source files either it is in the directory or in different directories to apply the tool on.
b. The, tool transforms the source code to transactions of itemsets.
c. EgyCD algorithm is applied to discover frequent itemsets in the source code that exceed a given frequency threshold.
d. The algorithm prunes all plagiarized text that appear completely in other plagiarized text to avoid duplicate results and report only original plagiarized not included in others.

Now a briefly description of how EgyCD algorithm works. Assume that T is the set of all source code statements, where each statement is considered a transaction. First, the algorithm starts by getting the first itemset F which is the set of all repeated statements in the source code. Then it initializes a counter i to 1. It also initializes a set CC equal to F where CC is a set will always contain all code clones discovered so far. Set CC_i is a sub set of CC always contains all code clones of length i while i increases for an iteration to the next. The second step is to do Cartesian product CC_i x F and store the results in CC_{i+1}. The third step is checking each item in the Cartesian product of length i + 1 against Apriori property which states that any subset of any frequent itemset should be frequent, to reduce the time of this check, only two subsets for any item in CC_{i+1} are checked, the first subset is equal to the same item in CC_{i+1} but after removing its first element, and the second subset is equal to the same item in CC_{i+1} but after removing its last element. If any of those two subsets is not frequent the item will be removed from CC_{i+1}. The fourth step is checking each item in the Cartesian product of length i + 1 to see if it exists in the set of all transactions T (i.e., the set of all source code lines in sequence) or not. If an item in the Cartesian product set exists as subsequence of transactions in T, then it is added to the code clones set, CC. Since the result of the Cartesian product can be massive, it is possible to generate the results on the fly in the memory without storing them and process them directly in the third step by checking their presence in the transactions. The fifth step is prune all code clones in CC of length i that exist in code clones of length i + 1. The fifth step is incrementing i by 1. The sixth step is trying to reduce the set F by pruning all items that didn't appear as a last item in any of code clones of length i . Finally the algorithm iterates over steps two to six until all items of the Cartesian product don't exist in any transactions. Below is the pseudo code of the algorithm.

```
1. T = set of all source lines
2. F = set of repeated source lines
3. CC = F
4. stillMore = true
5. i = 1
6. While (stillMore)
7. {
8.       stillMore = false
9.       CC_{i+1} = CC_i x F
10.      If i > 1 then
11.         CC_{i+1} = Check_Apriori(CC_{i+1},CC_i)
12.      End if
13.      For all e ∈ CC_{i+1}
14.      {
15.         if e ∈ T then
16.               add e to CC
17.               stillMore = true
18.         end if
19.      }
20.      prune CC by removing all e ∈ CC
         where |e| = i and e ⊂ S and S ∈
         CC where |S| = i+1
21.      i = i + 1
22.      prune all non used elements in F
23. }
```

Pseudo-code of EgyCD Algorithm

```
1.  Check_Apriori(CC_{i+1},CC_i)
2.  {
3.      For all e ∈ CC_{i+1}
4.      {
5.          a = all elements in e except
            first element
6.          if a ∉ CC_i then
7.              prune e from CC_{i+1}
8.          else
9.              b = all elements in e
                except last element
10.             if b ∉ CC_i then
11.                 prune e from CC_{i+1}
12.             end if
13.         end if
14.     }
15.     Return CC_{i+1}
16. }
```

Pseudo-code of `Check_Apriori(`CC_{i+1}`,`CC_i`)`

6. IMPLEMENTATION DETAILS

PD EgyCD algorithms were implemented as database applications using Adaptive Server SQL Anywhere version 11.0 with add on In-Memory version 11.0 and PowerBuilder version 11.5. This has multiple advantages. First, it perfectly matches the application of Apriori-based algorithms which are developed for mining databases. Second, the expressive power of SQL supports processing of transactions very easily and smoothly. Finally, PowerBuilder has powerful visualization capabilities that helped us visualize code clones in very simple ways and can also be upgraded with new views if needed.

6.1 Parallel Implementation

The main limitation in EgyCD is its speed and this comes from the Cartesian product process especially in detecting code clones of length 2, 3 and 4 [17, 19]. In this implementation, we speed up the calculation of the Cartesian product by dividing it over a specific number of threads. Before doing so, we need to tell each thread which elements of the first itemset it should deal with. Each item, i.e., a statement is stored in a database record and an integer field (item_thread) is added to the items table to indicate which thread is processing this item. Also we added a (clone_thread) field to the corresponding table to identify by which thread this code clone has been generated.

In Pseudo-code 1 of EgyCD algorithm, only the third line changes from:

CC = F

to:

CC = {x: x ∈ F and item_thread = thread_no}

This means that the thread only takes a subset of F to work on and use during its iterations to find larger frequent itemsets.

Simply each thread is responsible to get all code clones that starts by those items that he is responsible. To avoid a complex assembler routine that concatenate resultant code clones by each thread we relaxed the prune process of F set , if this process is not relaxed then each thread will produce a part of code clone not a complete code clone and then the assembler routine will be responsible in making a concatenate and merge among these code clones parts and it is so difficult to get the same number of code clones in case of using only one thread or the normal version.

Our experiments, as described in section 7, prove that parallelization on the same machine using multiple-threads can speed code clone detection by about 20%.

6.2 Distributed Programming Implementation

To further accelerate EgyCD, two distributed versions have been developed. The first version is a client-server implementation of EgyCD that can handle very large systems in a reasonable time by adding more machines as clients. The second version is an agent-based implantation of EgyCD. An agent is this context with an autonomous client that has its own database and does its work independently.

6.2.1 Client-server EgyCD

This version is similar to the parallel version in using multiple-threads. But in the client-server version we have two applications: the server application and the client, and one database server. The server application runs on the server machine and it is responsible of distributing the repeated items among clients. After that, it frequently checks whether all clients have finished their work. When they are all done, it calls the assembler and merger routines to concatenate code clones generated by different clients.

The following changes to EgyCD apply to both distributed versions of PD EgyCD. We added a field item_agent to the corresponding table to teach each client which lines/items it should deal with. We added clone_agent field to identify by which client/agent this code clone has been generated.

The client application works exactly like the parallel programming version except that the third line of the algorithm mentioned in section 5 will change again from:

CC = F

To:

CC = {x: x ∈ F and item_agent = agent_no

and item_thread = thread_no}

All the work related to the parallel programming implementation is included in the client application not in the server application.

6.2.2 Agent-based EgyCD

In the agent-based implementation, there are two applications: the server application and the agent application. An agent is nothing but a smarter client that works autonomously. Each agent has its own database. The Server application does the same work as in client-server version. It imports data and divides the items among the number of clients/agents. When the agent application starts it loads all the tables it needs from the server database and starts getting its solution as in the client-server version. After that, it sends the solution to the server application to integrate with other solutions. The main difference from the previous version is that once started, agents are autonomous and work independent from the server and its database.

7. CASE STUDIES.

EgyCD is capable of detecting clones of Types I, II and III [17]. Here, we are interested in evaluating the improvement and impact of using the parallel and distributed version. We have three cases study for detecting Type I.

7.1 Case Study I

The first case study shows what improvement in times results for using multi-threads. It is applied or a set of files. This is the example set of 25 C language files bundled with NICAD clone detector. We divided them into 5 groups; the first group contains 5 files and each consequent group contains the files of the pervious group and has 5 additional files, so the last group contains 25 files. The total size of these files is 332 KB and they collectively contain about 9180 lines of code.

The hardware used in this case is Intel® Core™ 2 Duo CPU E7200 Processor, 2.53 GHz, 2GB RAM, running windows XP.

Table 1shows the execution time for different numbers of threads (1,2, 3, 5).

Table 1: Results of Running Multi-Threaded EgyCD. Time is measured in seconds.

Seq.	LOC	Number of Threads			
		1	2	3	5
1	1915	0.96	0.84	0.72	0.72
2	4304	4.92	4.08	3.84	3.60
3	5949	8.40	6.36	6.00	6.00
4	7424	12.60	11.88	11.40	11.16
5	8454	17.64	12.72	14.64	14.88

Notice that increasing the number of threads up to specific number will not lead to decreasing the time, instead you will get increasing in time, this specific number depends on how many processors you have and the size of data and address buses.

An extensive comparison among normal EgyCD in which no threads are used is submitted in [17], so no need to make extra comparison especially that multi-threaded, client-server or agent-based version for related tools are very few and hence a difficulties in getting them for comparing arises.

Table 2 shows that an almost improvement of 20% on average is achieved in speed of code clone detection by using multiple threads to start with. But increasing the number of threads, on a single processor, does not improve data-mining based code clone detection.

Table 3 shows the total number of code clones detected using multi-threaded EgyCD. No differences in detecting number of code clones among different threads, this proves the correctness of using multi-threaded in which distributing the itemsets among threads. Really relaxing the pruning process of F set leads to this correctness otherwise in most cases it is impossible to get the same number of code clones as the normal version or one thread will produce.

Table 2: Percentage of Improvement in Time Efficiency by Using Multi-Threaded EgyCD.

Seq.	LOC	Number of Threads		
		2	3	5
1	1915	13%	25%	25%
2	4304	17%	22%	27%
3	5949	24%	29%	29%
4	7424	6%	10%	11%
5	8454	28%	17%	16%

Table 3: Total Number of Code Clones for Multi-Threaded EgyCD.

Seq.	LOC	Number of Threads			
		1	2	3	5
1	1915	80	80	80	80
2	4304	231	231	231	231
3	5949	345	345	345	345
4	7424	431	431	431	431
5	8454	486	486	486	486

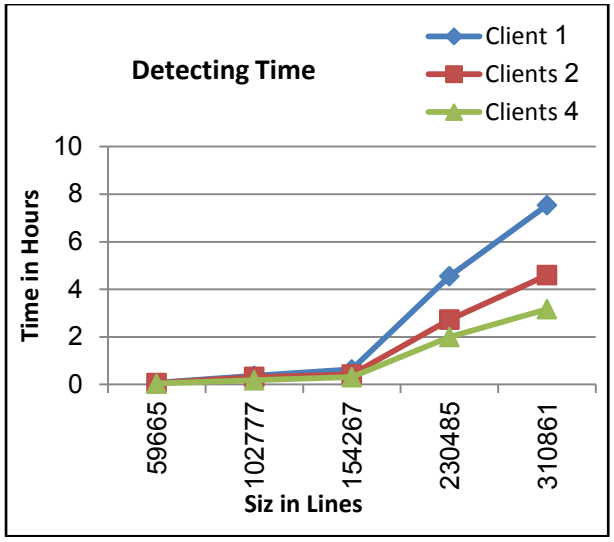
7.2 Case Study II

The second case study examines the increase in speed using client-server EgyCD. The third case study does the same for the agent-based version. The second and the third case studies is applied for very large scale to show that EgyCD can detect code clones for large scale systems, we randomly selected 2151 files from the JDK. Their size is 21.8 MB. The hardware used consists of 4 machines with Intel® Core™ i3-2310M Processor, 2.10 GHz, 4GB RAM, running windows 7 Professional and Sybase® Adaptive Server SQL Anywhere™ version 11.

In the second case study, client-server EgyCD was applied to selected subsets of the described data set, increasing in size, where each subset included the previous one and more source code. We used 1, 2 and 4 clients, each running on its own machine.

Table 4: Results of Running Client-server EgyCD. Time is measured in hours. % Column Shows the Percentage of Improvement in Execution Time.

Seq.	LOC	Number of Clients				
		1	2	%	4	%
1	59665	0.07	0.06	**14%**	0.04	**34%**
2	102777	0.37	0.32	**13%**	0.18	**52%**
3	154267	0.63	0.42	**33%**	0.31	**51%**
4	230485	4.55	2.73	**40%**	2.00	**56%**
5	310861	7.54	4.60	**39%**	3.17	**58%**
Average %				**28%**		**50%**



Graph 1. Results of Experiment with Client-server EgyCD.

Table 4 and Graph 1 show the results of the experiment taken for each source code subset for each number of clients.

7.3 Case Study III

The third case study shows the application of the agent- based distributed version of EgyCD on the same data sets used in case study II. Table 5 and Graph2 show the results of this experiment. As obviously expected, with the increase of the number of used agents, performance improves but not by the same ratio. This is due to the overhead of setting up each agent to work independently and then integrating the results.

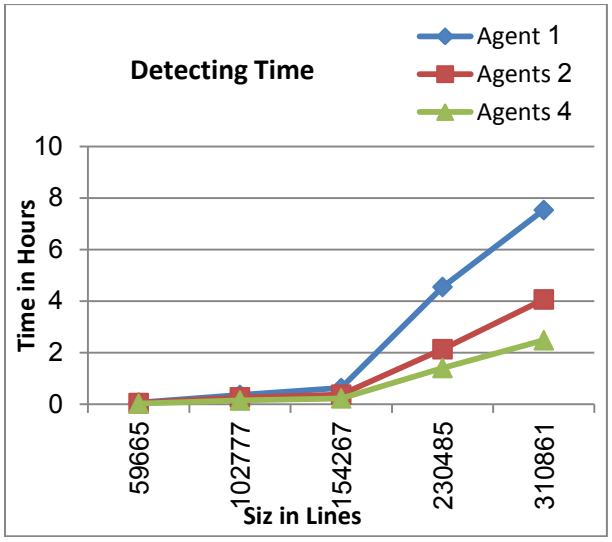
Table 5. Results of Running Distributed Agent-based EgyCD. Time is measured in hours. % Column Shows the Percentage of Improvement in Execution Time.

Seq.	LOC	Number of Agents				
		1	2	%	4	%
1	59665	0.07	0.05	**23%**	0.03	**59%**
2	102777	0.37	0.27	**26%**	0.15	**59%**
3	154267	0.63	0.37	**41%**	0.23	**63%**
4	230485	4.55	2.14	**53%**	1.41	**69%**
5	310861	7.54	4.07	**46%**	2.49	**67%**
Average %				**38%**		**63%**

Table 5 shows that using two agents instead of using one agent leads to decreasing the time by 38% percent on average and using four agents decreases processing time by 63%.

Graph 5 shows a graph comparison among the three agent system used, it is so clear that increasing the no of agents leads to decreasing the execution time in detecting code clones.

By comparing Tables 4 and 5, we see that agent-based EgyCD performs better by 26% on average. (average of reduction in time from client-server to agents version) This suggests that autonomous independent agents perform better than clients in data mining clone detection.



Graph 2. Results of Experiment with Agent-based EgyCD.

8. Analysis of the Results

In this section we analyze the results of our experiments and discuss the pros and cons of EgyCD. Our experiments showed that:

- Data mining techniques are suitable for code clone detection.
- Parallelism and distribution capabilities that are supported by modern database engines can be used to build fast distributed clone detectors.
- Using multi-threading on the same machine to speed up EgyCD reduced processing time by 20%.
- Using distributed multi-threading across multiple machines reduced processing time by about 63% at best when using 4 machines with agents.
- Agent-based clone detection is faster than client-based, i.e. having independent clients with separate databases.
- Aporiori-based algorithms and the representation of source code as items and clones as frequent itemsets can easily be distributed using distribution capabilities of modern database engines.
- With enough machines, code clone detection can be done at very large scale using these techniques.

On the other hand,

- Due to the exhaustive nature of Apriori-based algorithms, they can recover all code clones from a code base but slower than other code clone detection approaches.
- So, further non Apriori-based data mining techniques are worth of investigation in search for faster performance.

9. Conclusions and Future Work

In this paper, we presented new versions of EgyCD which is a data mining code clone detector. The first is a multi- threaded (parallel) version. The second and third are client- server and agent-based (distributed) versions.

Parallelism and distribution scale up data-mining-based clone detection to very large scale systems with reasonable time efficiency. We implemented the algorithms in a database-based language-independent family of clone detector tools called PD EgyCD. We presented a comparison with all versions of EgyCD to show the advantages and limitations of the new added features.

Future work will include the deployment of further data mining and non Apriori-based sequential pattern mining algorithms to further investigate the value of this family of algorithms in clone detection.

10. ACKNOWLEDGMENTS

Thanks to. Chanchal K. Roy, for his support, technical comments and research as well as his encouragement for this work, also thanks for Auni Ku and Ira for his support.

11. REFERENCES

[1] C. K. Roy, J. R. Cordy, R. Koschke, Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Comparison and Evaluation of Code Clone Detection Techniques, Science of Computer Programming, 74, 470-495, 2009.

[2] B. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the 2nd Working Conference on Reverse Engineering, WCRE 1995, pp. 86-95, 1995.

[3] C. K. Roy and J. R. Cordy, An Empirical Study of Function Clones in Open Source Software Systems. In Proceedings of the 15th Working Conference on Reverse Engineering, WCRE 2008, pp. 81-90, 2008.

[4] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner. Do Code Clones Matter? In Proceedings of the 31st International Conference on Software Engineering (ICSE'09), pp. 485–495, Vancouver, Canada, May 2009.

[5] J. H. Johnson. Identifying Redundancy in Source Code Using Fingerprints. In Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON' 93), pp. 171–183, Toronto, Canada, October 1993.

[6] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In Proceedings of the Second Working Conference on Reverse Engineering(WCRE'95), pp. 86–95, Toronto, Ontario, Canada, July 1995.

[7] A. Chou, J. Yang, B. Chelf, S. Hallem and D. R. Engler. An Empirical Study of Operating System Errors. In Proceedings of the 18th ACM symposium on Operating systems principles (SOSP'01), pp. 73–88, Banff, Alberta, Canada, October 2001.

[8] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. IEEE Transactions on Software Engineering, 32(3):176–192, 2006.

[9] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[10] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo, Comparison and Evaluation of Clone Detection Tools, Transactions on Software Engineering, 33(9):577-591, 2007.

[11] Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. Transactions on Software Engineering, Vol. 28(7): 654-670, July 2002.

[12] Chanchal Kumar Roy and James R. Cordy, A Survey on Software Clone Detection, Technical Report No. 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, September 26, 2007.

[13] Raghavan Komondoor and Susan Horwitz. Using Slicing to Identify Duplication in Source Code. In Proceedings of the 8th International Symposium on Static Analysis (SAS'01), Vol. LNCS 2126, pp. 40-56, Paris, France, July 2001.

[14] Agrawal, R. and Srikant, R. 1995. Mining sequential patterns. In Eleventh International, Conference on Data Engineering, P. S. Yu and A. S. P. Chen, Eds. IEEE Computer Society, Press, Taipei, Taiwan, 3-14.

[15] Chao Liu, Chen Chen, Jiawei Han and Philip S. Yu. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06), pp. 872-881, Philadelphia, USA, August 2006.

[16] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based Code Clone Detection: Incremental, Distributed, Scalable. Int. Conf. Software Maintenance (ICSM), 2010.

[17] A. Matarawy, M. El-Ramly and R. Bahgat. Code Clone Detection Using Data Mining, Conference of Institute of Statistical Studies and Research (ISSR), Cairo University. (to appear in Dec. 2012).

[18] S. Livieri, Y. Higo, M. Matushita, K. Inoue, Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder, Graduate School of Information Science and Technology, Osaka University1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan, 2007

[19] Vera Wahler, Dietmar Seipel, J¨urgen Wolff v. Gudenberg, and Gregor Fischer. Clone Detection in Source Code by Frequent Itemset Techniques, Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on16-16 Sept. 2004.

[20] M.-S. Chen, J. Han, and P. S. Yu. Data mining: an overview from a database perspective. IEEE Trans. On Knowledge And Data Engineering 8, 866-883,1996.

[21] Q. Zhao, S.S. Bhowmick, Sequential pattern mining: a survey, Technical Report Center for Advanced Information Systems, School of Computer Engineering, Nanyang Technological University, Singapore, 2003.

[22] Jiawei Han, Micheline Kamber: Data Mining – Concepts and Techniques, Kaufmann, 2001.