Parallel Testing in Behavior Driven Development

Felipe S. de Amorim
Federal Institute of São Paulo
Capivari - SP, Brazil
felipe.murin@gmail.com

Lincoln M. Costa
Computer Systems Engineering Program, Federal University of Rio de Janeiro
Rio de Janeiro - RJ, Brazil
costa@cos.ufrj.br

Rodolfo Adamshuk Silva
Department of Software Engineering, Federal University of Technology - Paraná
Dois Vizinhos - PR, Brazil
rodolfoa@utfpr.edu.br

Francisco Carlos M. Souza
Department of Software Engineering, Federal University of Technology - Paraná
Dois Vizinhos - PR, Brazil
franciscosouza@utfpr.edu.br

ABSTRACT

The testing process consists of activities that demand efforts as producing, executing, and validating test scenarios. Covering all test scenarios manually is unfeasible since it is error-prone and labor-expensive. Thereby, partial or complete automation reduces costs and increases tests' effectiveness. The increasing availability of hardware resources provides opportunities to scale testing using parallel execution of test cases or suites blocks. Some tools perform parallel execution of tests, but their use requires complicated settings, and when combined with some methodologies as Behavior-Driven Development, it may create an overhead for users. This paper presents the Multi-Threaded Testing (MTT) tool for parallel execution of test scenarios in the context of Behavior-Driven Development that aims to reduce the computational time required to test Java projects. Furthermore, the present paper reports an experimental study to evaluate the MTT tool's performance in two different hardware configurations. Our results demonstrate the MTT reached a speedup of 4,59 using ten threads in CPU Intel Core i5-9300H with an efficiency of 46%, and a speedup of 3,45 with an efficiency of 43% using eight threads in CPU Intel Core i7-7700HQ.

KEYWORDS

Software testing, Parallel testing, BDD

1 INTRODUCTION

Software testing is a fundamental activity to achieve and assess the quality of software. The testing activity refers to the generation and execution of a test data set and analyze the program behavior for fault detection. Given its importance, in 2020, software development companies committed around 22% of the project's budget in quality assurance activities [1]. This reality reflects how companies are dealing with testing activities, applying them during all software life-cycle. Around 52% of the companies plan and execute tests as early as possible during the software development process [1].

The software testing process is time-consuming, expensive, and error-prone due to its complexity in identifying scenarios in which the program is more likely to fail. Besides that, manual testing is practically impossible due to the high quantity of test scenarios used for small programs. Consequently, partial or complete automation reduces costs and increases the effectiveness of tests [2]. Automated testing is the solution to ongoing testing throughout the software development life-cycle while maintaining the tested artifacts' quality and assertiveness. Hence, its maximization was encouraged in 51% of the companies in 2020 [1].

The agile development processes that support continuous integration improves continuous testing, converting it into an ever-growing activity. For instance, a new software increment may affect the system's features, therefore it requires integrated testing. The testing automation activity is inherent to software development's agile practices and generally employs unit tests to validate incrementally small functional requirements. The testing automation is present in techniques such as Test-Driven Development (TDD), Acceptance Test-Driven Development (ATDD), and Behavior Driven-Development (BDD). Moreover, testing automation's success relies on automation tools [3].

There are many testing tools to support this activity's essential process. The testing automation consists of writing code that creates a test scenario, executes an action, and validates the results according to the software specification. The Selenium is one of the most popular tools to support functional testing (i.e., testing based on system specification/requirements) at unit and integration levels. The Cucumber is a platform to support the BDD technique in Java projects. It uses the Gherkin syntax to describe each feature's scenarios and supports the automation of tests based on data defined in each scenario.

In software testing, the number of artifacts (e.g., codes, features, class integration) impacts the number of test cases. The software may have several features to implement and, for each one, the creation of several scenarios is required to assure its quality. For instance, around fifty test scenarios are executed for programs with a few features. Therefore, for programs with a high number of complex features, around five thousand scenarios must be performed. Consequently, the time necessary to execute them may be hours or days for more complex software.

As a solution, the tester may apply test case prioritization techniques. According to Rothermel et al. [4], the prioritization of test cases is a complex task due to the high number of parameters (or factors) that may be considered in the prioritization, and the constant evolution of the software may require the inclusion and exclusion of test cases, changing the prioritization order. Another solution consists of divide test cases or suites in blocks for parallel execution [5]. Some tools support parallel execution as Selenium, Pabot,

and JMeter Parallel Controller. However, these tools' complexity hampers their use in contexts such as BDD. Besides that, the time required to set up these tools may prevent their use.

This paper introduces the Multi-Threaded Testing (MTT) tool for parallel execution of test scenarios in the context of Behavior Driven-Development methodology. MTT aims to reduce the computational time required to test Java projects that uses Cucumber as support. Cucumber is the current state-of-the-art software testing platform for BDD. MTT tool allows the tester to configure the parallel execution of test scenarios by choosing the number of threads to execute. This configuration does not require new test cases or already automated test cases. MTT tool uses the test information defined by the tester and parallels the test methods' invocation. The configuration is simple to adapt to include new features and test scenarios.

Overall, the contributions of the present work can be summarized into the following points: *(i)* an automated test case execution tool that uses multi-thread programming; *(ii)* a performance evaluation of the proposed tool in a testing scenario context.

The remainder of this paper is organized as follows: Section 2 details the background and related work about Behavior-Driven Development and parallel testing. Section 3 details the automated tool proposed. Section 4 describe the performance evaluation of the tool. In Section 5 the conclusions and future directions are discussed.

2 BACKGROUND

In the agile process of software development, the coding and deployment of new functionalities are constant. Commonly, the deployment of a new patch affects the system's existing features. Hence, testing efforts increase regularly in each deployment because tests should cover the system entirely. Agile development methodologies increase the necessity of testing automation throughout the development process. Automated tests simulate the end-user activities, but a script reproduces them. Those scripts may be written in a wide variety of programming languages.

According to A. Contan [6], automated tests should be performed on three levels: Unit, Integrated, and User Interface. Unit tests verify an isolated functionality of the software, and it is the base of the Test-Driven Development (TDD). It is used to validate components such as a class or a function. The integrated testing validates interactions between components regarding functionality or requirement, and it is used in Behavior-Driven Development (BDD). Finally, user interface testing validates the software's behavior from the end-user perspective. It verifies characteristics such as displaying elements, responsiveness, and interactions. Tests that involve users or the client are used on Acceptance Test-Driven Development (ATDD).

2.1 Testing in Behavior Driven-Development

BDD is a set of practices to guide the development of software in which the developer explores the desired system behavior with examples in conversations with the team and formalizes them into automated tests to guide the development [7]. In the context of Agile development in Java projects, Behavior Driven-Development may be applied in projects of all sizes. Most tools associated with BDD help the developer formalize and automate parts of BDD practices. To support the development of Java projects in BDD, Cucumber is the best-known platform that guides the development.

BDD does not use abstract specifications though it uses examples to explore the behavior that must-have in the system. The common flow of development in a Java project using BDD with Cucumber starts with creating a file *.feature* which describes the scenarios using the Gherkin syntax. A scenario represents a user story of a functional feature of the system. The Gherkin uses the keywords *Given*, *When*, *And* and *Then* to describe steps to be executed in a feature scenario. Figure 1 presents an example of a hypothetical behavior using the Gherkin syntax. Therefore, given that the user is on google.com, when the user inserts a value in the page's text field, then the value shown must be the same.

```
Scenario: TC_01
  Given User is on google
  When User types thread number
  Then The type is validated
```

Figure 1: Example of a scenario on a *.feature* file

After defining the scenario, the last step consists of the implementation of the test scenarios. A scenario is defined using "steps" to guide the execution. Each step is a method that follows what was defined in the *.feature* file. Those steps generally are tagged with Gherkin syntax. The *@Given* indicates that that block is running a precondition. *@When* is where the user actions are described to fulfill the scenario objective. In *@Then*, the assertions validate the scenario results. The tag *@And* may be used along with any other to include multiple lines within that section. Other testing tools may be used to support the execution as Selenium to interact with the web interface and JUnit or TestNG to make assertions about the result. Figure 2 presents the implementation of the behavior shown in Figure 1. Each step is a method that uses the Selenium WebDriver tool to interact with the web page and TestNG to make assertions. In this testing scenario, the script will open a web browser, insert a number in the variable thread number, and validate if this number is the same as the text field.

```
@Given("User is on google")
public void userIsOnGoogle() {
    driver.navigate().to( s: "https://www.google.com/");
}

@When("User types thread number")
public void userTypesThreadNumber() {
    driver.findElement(By.xpath(googleInput)).sendKeys(threadNumber);
}

@Then("The type is validated")
public void theTypeIsValidated() {
    String fieldText = driver.findElement(By.xpath(googleInput)).getText();
    assertEquals(threadNumber, fieldText);
}
```

Figure 2: A scenario implementation on a Java class

The last step is to execute the tests, and it may be done in two different ways. The first uses a Java class with JUnit's *@CucumberOptions* annotation. In this method, the tester may create several Runner classes with different predefined configurations, allowing easier command line calls. In the second method, a Cucumber Java build configuration called Runner is created to execute the scenarios directly through the *.feature* files. Figure 3 presents an example of a Runner class with *@CucumberOptions* annotation configuration. Both methods need at least two parameters, the *features*, and the *glue*. The *features* parameter indicates the folder with the *.feature* files containing the scenarios to be executed. The *glue* parameter indicates the package with the Java classes containing the methods related to each scenario step.

```
@RunWith(Cucumber.class)
@CucumberOptions(
        features = "src/test/resources/products/",
        glue={"com/mtt/products"}
)

public class Runner {

}
```

Figure 3: Example of a Java class using *@CucumberOptions*

2.2 Parallel Testing

The testing activity is generally done sequentially, with test blocks' execution one after another. Therefore, with the increased use of multi-core processors, concurrent programs may be a solution to improve performance and a way to explore the hardware resources available on the computer. Parallel testing consists of the execution of testing tasks (units or blocks) concurrently [8]. The objective is to increase the throughput and efficiency of test execution. Rivoir [9] showed the advantages of using parallel execution to solve the high cost related to the execution of large sets of test cases.

Waivio [10] presents three types of parallelism in the automation of testing units. A testing unit is composed of several testing functions. For example, a program with two features may have a testing unit for each feature. Each unit may have several test cases that cover different scenarios. Type I parallelism is defined as testing multiple Units Under Test (UUT) in parallel. In this type of parallelism, testing units are decomposed and mapped regarding the number of resources available (processors or cores). In this case, several units are executing in parallel, and each one executes different test cases sequentially. Type II parallelism (or Vector Test Program Set [11]) is defined as running multiple tests on a single UUT in parallel. It may be useful when the coupling between test scenarios is weak and prerequisites among them are nonexistent. Finally, type III parallelism is the parallel execution of a single test case's test steps or actions. This type of execution requires a design to identify which part of the test may be executed in parallel observing data dependency.

Bazylevych and Franko [12] review the efficiency of parallel path analysis based on the Control Flow Graph (CFG) to generate unit tests. In this approach, source codes in the C programming language are parsed to Abstract Syntax Tree, and then a CFG is created. After that, the graph is interpreted as a tree, and all paths of this tree are recorded. The efficiency in the execution of this approach depends on the number of processors and the amount of time need to follow one path. The authors conducted an experiment to analyze the time, speedup, and path parallelization execution efficiency. As a result, they showed that the proposed approach's efficiency is near 80% when executed in four processors.

Some tools support the parallel execution of testing, as JMeter, Selenium, and Pobot. JMeter [1] is a performance testing framework created by Apache, and it has been widely employed as a performance testing tool for Web applications. JMeter provides a parallel controller that allows concurrent sampling by many threads, which helps perform different testing scenarios. The parallel controller works through a set of elements called Thread Group. These elements are used to define the number of threads, ramp-up period, and the number of repetition of the test. Each thread is responsible for simulating a user and execute a test suite simultaneously and independently of other threads. In some cases, automation is challenging since it requires various parameters to specify, mainly for complex applications that use dynamic content. Lastly, the load testing using JMeter may use a high quantity of memory, leading to overload and no guarantee of the testing.

Selenium is a set of three tools used to test web applications: Selenium IDE, Selenium WebDriver, and Selenium Grid. The Selenium supports test automation on web browsers such as Firefox, Internet Explorer, Google Chrome, Safari, and Opera [13]. To run Selenium WebDriver tests in parallel, it is necessary to set up the Selenium Grid Server as a Hub. Each node in the hub will run the test in a different web browser. To run tests in parallel, TestNG [2] unit testing framework is used to create tests that may be executed in parallel. It uses a multithreading model to run tests in parallel. This approach's problem is the necessity to use the TestNG or another testing framework and create a new configuration file in XML (eXtensible Markup Language). The configuration is complex, and it is time-consuming because if a test case is included or excluded, it is necessary to adapt the configuration file.

Pabot [3] is a tool that allows the parallel execution of tests for Robot Framework [4] - a test automation framework. Pabot allows the tester to split one execution into multiple processes that run concurrently and provides keywords that help communication and data sharing between the executor processes. The problem with this tool is that it cannot be used in BDD using the Cucumber platform.

The tool presented in this paper aims to automatically distribute test scenarios within new threads with no need to re-write test cases or modify already automated test cases, gathering the test instructions and invoking the test methods.

[1] https://jmeter.apache.org/
[2] http://testng.org
[3] https://pabot.org/
[4] https://robotframework.org/

3 MULTI-THREADED TESTING TOOL

The Multi-Threaded Testing (MTT) tool aims to reduce the computational cost required to execute a set of test scenarios (units) in the context of Java programs using the Cucumber platform for Behavior Driven-Development projects. MTT allows the tester to execute a project's test scenarios in parallel using multithread execution of each test scenario.

3.1 MTT Configuration

In a Java project using the Cucumber platform, the testing process using BDD follows the development flow defined in Section 2.1. At the end of this process, the project commonly has a package with all *.feature* files. Each file may contain scenarios that represent user stories of the system. Scenarios are written using Gherkin syntax that allow the tester to define how the system should behave through steps (*Given*, *When*, *And* and *Then*). Besides that, for each feature, there is a package with *.java* files containing all test methods that implement each step defined in each scenario. Those methods may use other tools and plugins as JUnit, TestNG, and Selenium to simulate a web browser's execution and make assertion between values. Finally, the Runner class contains the configuration necessary to execute the Cucumber project and the test scenarios.

The MTT tool is composed of two classes called MTT and Core (shown in Figure 4) that need to be imported into the project before its use. MTT class manages the process of reading the files and creating the threads. The Core class encapsulates all the methods that implement the scenarios class (Figure 2). Therefore, Selenium directives such as web drivers and all the browser actions are unnecessary.

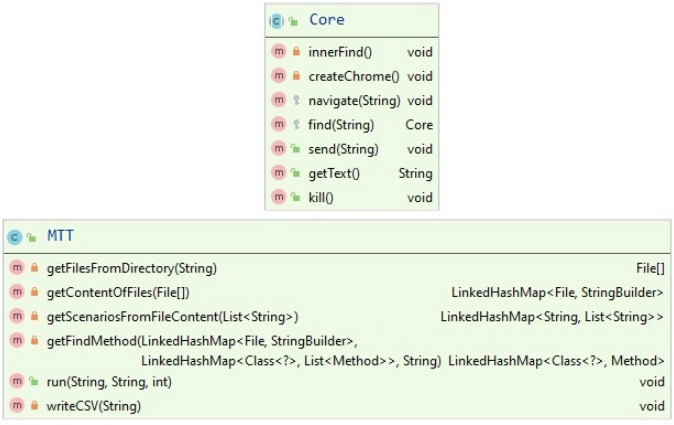

Figure 4: Class diagram of MTT tool

After that, it is necessary to change the Runner class and instantiate the MTT class. The tool set up is composed of a call to the run method with three parameters: (1) the path to the *.feature* file directory, (2) the path of the *.java* class directory, which contains the step methods implementation, and (3) the number of threads to execute. This configuration is similar to the cucumber configuration of runner (shown in Figure 5).

This version of the runner class does not use Cucumber tags as *@RunWith* and *@CucumberOptions*. Instead, JUnit or TestNG tags

```
public class Run1 {
    private int maxInstances = 1;

    @Test
    public void runnable1(){
        new MTT().run(
                stepDefs: "/src/test/java/suites/mtta/stepdefs/",
                features: "/src/test/java/suites/mtta/features/",
                maxInstances
        );
    }
```

Figure 5: Runner class for MTT tool

such as *@Test* are used to create test methods. Those methods use as parameter the same *@CucumberOptions* arguments as shown in Figure 3, plus the number of threads represented by the variable *maxInstances*.

An advantage of this new runner class is the possibility to organize different executions. For instance, in the Cucumber options, a configuration is made per class. In MTT, however, as they are called method via *@Test* tag, it is possible to have several classes with different configurations. Besides that, it is possible to have several classes with Cucumber options.

3.2 MTT Execution Flow

The MTT tool is composed of 3 steps: (1) Scenarios reading, (2) Test reading, and (3) Parallel execution. Figure 6 presents an overview of the MTT tool.

3.3 Step 1 - Scenarios Reading

The first step consists of the reading of all test scenarios defined in the *.feature* files. As seen in Section 2.1, each feature file may contain several test scenarios. Each scenario is written using Gherkin syntax, and each keyword (*@Given*, *@When*, *@Then*, and *@And*) represents a step to be executed in the scenario.

Further, each scenario may contain several test inputs (test data). Therefore, the MTT tool reads all *.feature* files of the project and save the scenarios and test inputs. As a result of this step, a list with all scenarios is created to be used in Step 3.

3.4 Step 2 - Test Reading

The second step consists of the reading of all test methods implemented. As seen in Section 2.1, the test scenarios are implemented in a class with methods that execute each step defined in the *.feature* files. In this step, all methods are read and saved in a list to be used in Step 3.

3.5 Step 3 - Parallel Execution

The last step is the parallel execution of different test scenarios. In the configuration runner class, the tester defines the number of threads to be executed, and the MTT tool creates the threads that will execute all the scenarios. For example, if the tester sets the number of threads as 3, then three threads will execute the test scenarios following the information collected in the *.feature*

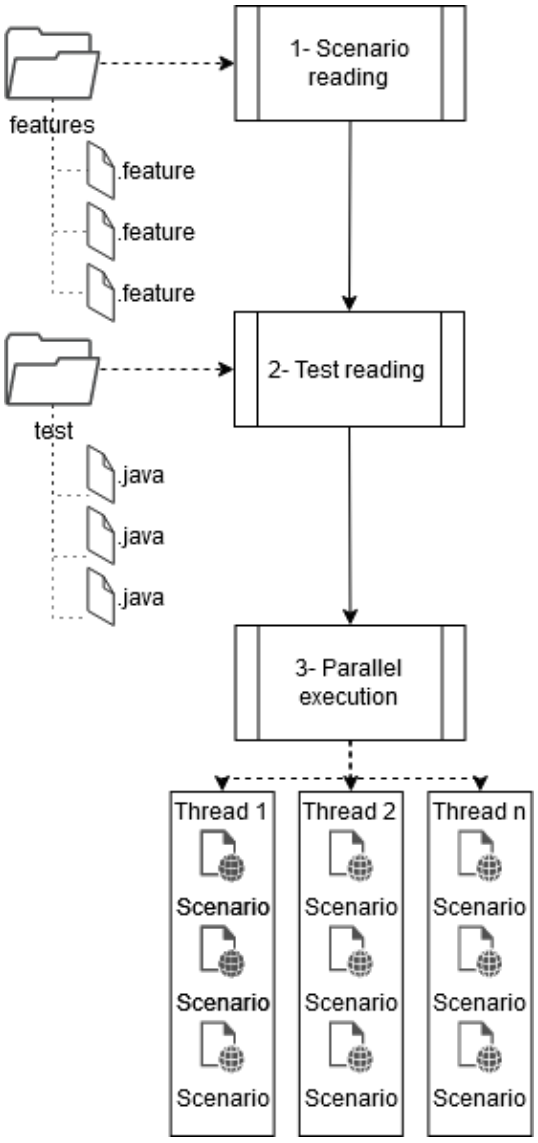

Figure 6: Overview of the MTT

and *.java* classes and saved in a list as shown in steps 1 and 2. As a scenario is composed of several steps, they will be executed sequentially. Furthermore, as a scenario may have several test inputs, each test input will be executed sequentially. This execution model represents type II of test parallelism described by Waivio [10].

The MTT tool implements a concurrent programming model called thread pool. When a thread finishes the execution of one scenario, it looks back at the list of scenarios to be executed and selects the next scenario to be executed until there are no more scenarios to be executed. Figure 7 presents the thread pool model.

4 PERFORMANCE EVALUATION

The execution of automated tests may lead to performance problems once test scenarios must be executed sequentially. MTT tool brings a solution to this problem and allows the tester to parallelize test scenarios' execution, increasing the testing performance.

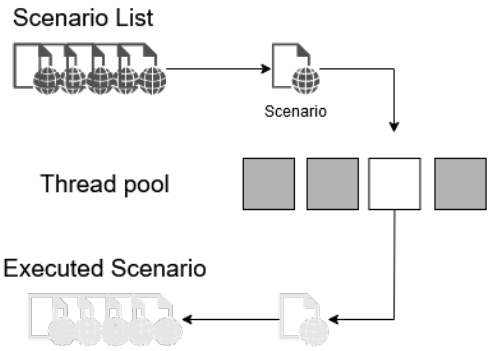

Figure 7: Thread pool model

This section presents a study to evaluate the MTT tool's performance in three different hardware configurations. The objective is to identify the speedup gained using different threads for the same workload.

4.1 Evaluation Setup

In this performance evaluation, it is intended to evaluate the performance of MTT tool in two different hardware setups:

- Laptop Asus Predator Helios 300 with an Intel Core i7-7700HQ CPU @2.80GHz, 16GB of RAM, and a Seagate ST2000LM007 2TB 128MB Cache SATA 6.0Gb/s;
- Laptop Avell G1555 MUV with an Intel Core i5-9300H CPU @2.4GHz, 16GB of RAM, and a ADATA SX8200PNP 512GB HD 500MB/s;

Windows 10 was used for all hardware setups as an OS, and the software used for the execution are IntelliJ Community IDE version 2019.3.1, JDK version 1.8.0.231, Selenium version 3.141.59, and TestNG version 6.14.3. The Selenium plugin is responsible for the navigation on the web driver (browser), and the TestNG plugin is responsible for executing the test suite.

The program used in this study is a web application for triangles classification. Triangles are classified depending on the relative sizes of their elements. As regards their sides, triangles may be Scalene (all sides are different), Isosceles (two sides are equal), or Equilateral (all three sides are equal). This project has a feature that consists of the triangle classification considering the size of their side. Therefore, three testing scenarios were created to verify a possible triangles class. For each scenario, thirty test cases were used with different input values.

4.2 Implementation and Measurement

This study considers two factors with some levels (or treatments). The first level is the hardware setup and has two levels in CPU: Intel Core i7-7700HQ, Intel Core i5-9300H. The second factor is the number of threads with twelve treatments: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, and 20. The design defined for this study is a Full Factorial Design with Replication. A full factorial design utilizes every possible combination at all levels of all factors. Table 1 presents the design of the study divided into three categories described in

the first column. The second column presents the levels of factor 1 (CPU). The third column represents the second factor, and the last column presents the third factor.

Study scenario	CPU	Thread number
1	Intel Core i7-7700HQ	1 to 10 15 20
2	Intel Core i5-9300H	1 to 10 15 20

Table 1: Design of the study.

This study uses the turnaround time that represents the time between the start of the execution until its end. The execution time was obtained in the IntelliJ IDE and collected for each repetition. Ten repetitions were performed, and the average of the time was calculated.

In the evaluation of a parallel system, the aim is to find how much performance gain is obtained by the parallelization of an application compared to the sequential version. The Relative speedup compares the parallel version on one processor compared to a number p of processors [14]. The relative speedup was calculated by the ratio between the time on one thread and the time on using each treatment of treads.

Efficiency measures the time a processor is used and is defined by the ratio between the speedup and the number of processors. The efficiency was calculated by the ratio between the speedup by the number of threads, using the formula presented in Grama et al. [14].

In an ideal parallel system, the speedup is equal to the number of processors and the efficiency is equal to 1. However, in practice, the speed increase is less than the number of processors and the efficiency is between 0 and 1, depending on the effectiveness of the processors [14]. Therefore, in our experiment even if the speedup reached is high, if it uses several threads in parallel, the efficiency will be low.

4.3 Results

The results of the experiment are presented in Table 2 in which the first column presents the CPU; the second column shows the thread number, the third column exhibits the time in minutes (average), the fourth column displays the speedup, and the last column shows the efficiency reached.

The Figure 8 shows the executions' speedup. It is possible to observe a sublinear speedup once the speedup obtained is less than the number of threads used in the execution. The critical point for this study is between eight and ten threads. After this point, the speedup and efficiency decrease. This decrease is understandable once the increase in the number of threads increases the JDK management considering resources and processing order. After ten threads, the overhead generated by communication and synchronization intrinsic to concurrent programs starts to overcome the processing time's gaining.

CPU	Thread	Minute	Speedup	Efficiency
Intel Core i7-7700HQ	1	8,35	1,00	100%
	2	4,72	1,77	88%
	3	3,95	2,12	71%
	4	3,11	2,69	67%
	5	2,64	3,16	63%
	6	2,52	3,32	55%
	7	2,51	3,33	48%
	8	2,42	3,45	43%
	9	2,44	3,42	38%
	10	2,43	3,43	34%
	15	2,91	2,87	19%
	20	4,66	1,79	9%
Intel Core i5-9300H	1	8,58	1,00	100%
	2	4,61	1,86	93%
	3	3,38	2,54	85%
	4	2,89	2,97	74%
	5	2,60	3,30	66%
	6	2,19	3,91	65%
	7	2,05	4,18	60%
	8	2,01	4,27	53%
	9	1,96	4,38	49%
	10	1,87	4,59	46%
	15	2,11	4,06	27%
	20	2,68	3,21	16%

Table 2: Design of the study.

In the Figure 8, it is possible to see that the speedup of CPU Intel Core i5-9300H is higher than CPU Intel Core i7-7700HQ. These CPUs are from different generations: the former is from the ninth generation, and the letter is from the seventh generation. Even with the same number of cores (four) and the same number of threads (eight), i5-9300H has improvements such as Max Turbo Frequency of 4.10 GHz and 8 MB Intel Smart Cache. Other hardware characteristics such as RAM may influence the speedup Even with a higher speedup, the critical point of the study is ten threads for both CPUs.

This study's objective was to observe the speedup of the MTT tool. Therefore, different thread numbers were used to execute one testing workload of 120 test cases. As a result, we identified that the higher speedup was 4,59 using ten threads in CPU Intel Core i5-9300H with an efficiency of 46%. In CPU Intel Core i7-7700HQ, the higher speedup was 3,45 with an efficiency of 43% using eight threads.

4.4 Threats to Validity

Construct validity refers to the relation between theory and observation [15]. Threats to construct validity regard the extent to which the setting reflects the construct under study. A threat in this study is the thread number. It directly influences the execution time and consequently in the speedup and efficiency. Twelve thread numbers were used to mitigate this threat, ranging from 1 to 10,

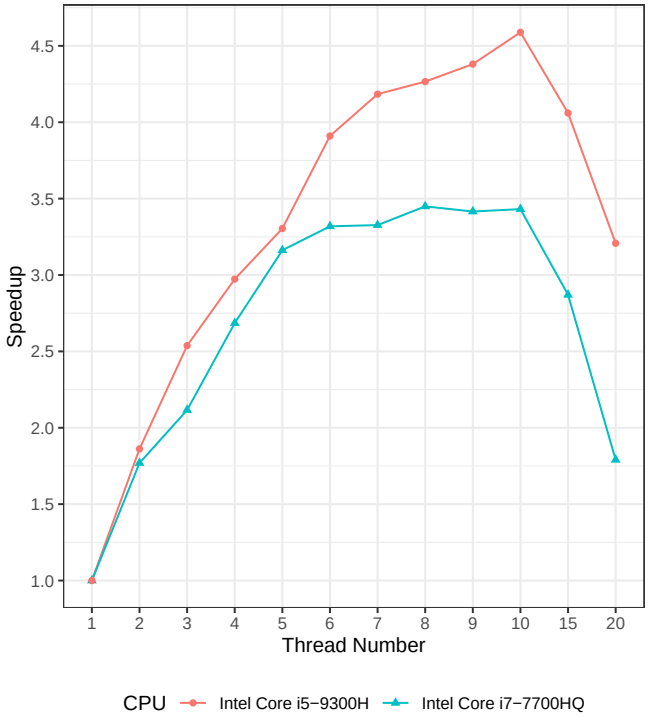

Figure 8: Speedup of the results

15, and 20. Besides that, ten repetitions were performed for each thread number.

External validity refers to the generalization of the results [15]. Threats to external validity concern our ability to generalize the results outside the study setting. A threat in this study is the representativeness of the program. A web application available online was used with 120 different test cases to mitigate this threat. Two different hardware configurations were used to examine the speedup and efficiency difference.

4.5 Discussions

The primary objective of the MTT tool is to support the parallel execution of test scenarios in the context of BDD, as described in Section 3. However, due to its possibility to create several test instances, it can be used in performance testing. One functionality may be tested until its limit to observe how much simultaneous access this functionality can support.

The MTT does not demand a structure to test cases and scenarios. Therefore, it may be adapted to other contexts beyond BDD. The instantiation of MTT to parallelize test scenarios in BDD came from the authors' real necessity to improve the test automation process in a context with several test scenarios to be executed. As future work, a framework structure will be available for testers that want to instantiate MTT to apply it in different agile methodologies.

Besides the support for web application testing, MTT can be used in mobile tests to parallelize scenarios within devices. The tester must configure the devices to be used, and each one will be executed in one thread. Therefore, the number of threads executing in parallel will be the same as the number of devices. This new functionality has some limitations once the tester cannot analyze the simultaneous test's correct impact without closing and opening devices manually; therefore, more studies are necessary for this type of testing.

MTT tool has a limitation regarding the ideal configuration considering the number of threads. It is known that only the increase in the number of threads does not necessarily mean an increase in the speedup and sometimes can become the program worse than the sequential. Consequently, the tester must consider a previous analysis of its hardware configuration and the test workload to be executed and seek to define the best configuration.

Considering the types of parallelism defined in Waivio [10], the MTT tool uses the thread pool model of execution that represents type II of test parallelism. The MTT tool is similar to JMeter and Pabot, once it is possible to set the number of threads to be executed. However, the MTT has the advantages of allowing parallelization in the context of BBD projects and has an easy configuration setup.

5 CONCLUSION

Commonly, creating and executing tests to ensure product quality is considered a complex activity. The growing number of artifacts (codes, features, class integration, among others) for testing directly influences the number of test cases and, consequently, its time to be executed. To address this problem, parallel execution of test scenarios is promising.

MTT (Multi-Threaded Testing) is a tool to parallelize the execution of test scenarios in Java projects using BDD (Behavior-Driven Development) and Cucumber. The tool aims to reduce the time required to execute testing scenarios by multi-thread test automation execution.

A study was conducted to evaluate the performance of MTT using two different hardware configurations and twelve thread numbers. The results showed that MTT could reach a speedup of 4,59 using ten threads in CPU Intel Core i5-9300H with an efficiency of 46%, and a speedup of 3,45 with an efficiency of 43% using eight threads in CPU Intel Core i7-7700HQ. This scenario may hugely increase the test response and impact during a software development life-cycle.

Future work will involve a deeper understanding of resource peak analysis. Besides that, new studies are necessary to understand how to select the ideal number of threads for each context. Finally, mobile testing functionality must be validated.

6 ACKNOWLEDGMENT

The authors would like to thank Adriane Kaori Oshiro and Lucas Daolio for comments that greatly improved this work, and we thank Marcelo Roland Bernardino, Denis Augusto L. de Castro and Tiago Barbosa Wenceslau for their insights.

REFERENCES

[1] Capgemini. World quality report 2020-21. https://www.capgemini.com/research/world-quality-report-wqr-20-21/, 2020. Acesso em: 04 de dezembro de 2020.

[2] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004. ISBN 0471469122.

[3] Ida Bagus Kerthyayana Manuaba. Combination of test-driven development and behavior-driven development for improving backend testing performance. *Procedia Computer Science*, 157:79 – 86, 2019. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2019.08.144.

[4] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, 1999. doi: 10.1109/ICSM.1999.792604.

[5] E. Starkloff. Designing a parallel, distributed test system. In *2000 IEEE Autotestcon Proceedings. IEEE Systems Readiness Technology Conference. Future Sustainment for Military Aerospace (Cat. No.00CH37057)*, pages 564–567, 2000. doi: 10.1109/AUTEST.2000.885641.

[6] L. Miclea A. Contan, C. Dehelean. Test automation pyramid from theory to practice. In *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2018.

[7] R. Lawrence and P. Rayner. *Behavior-Driven Development with Cucumber: Better Collaboration for Better Software*. Pearson Education, 2019.

[8] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy. Automatic generation of parallel unit tests. In *2013 8th International Workshop on Automation of Software Test (AST)*, pages 40–46, 2013. doi: 10.1109/IWAST.2013.6595789.

[9] J. Rivoir. Parallel test reduces cost of test more effectively than just a cheap tester. In *IEEE/CPMT/SEMI 29th International Electronics Manufacturing Technology Symposium (IEEE Cat. No.04CH37585)*, pages 263–272, 2004. doi: 10.1109/IEMT.2004.1321674.

[10] Nathan Waivio. Parallel test description and analysis of parallel test system speedup through amdahl's law. In *2007 IEEE Autotestcon*, pages 735–740. IEEE, 2007.

[11] C. McNatt and T. Gaudette. Vectorized test program sets using matlab and the teradyne ai-710 analog test instrument. In *2006 IEEE Autotestcon*, pages 432–437, 2006. doi: 10.1109/AUTEST.2006.283701.

[12] Roman Bazylevych and Andrii Franko. Parallelization of unit tests generation by control flow graph analysis. In *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*, volume 2, pages 161–164. IEEE, 2019.

[13] U. Gundecha. *Selenium Testing Tools Cookbook*. Community experience distilled. Packt Publishing, 2015. ISBN 9781784392512. URL https://books.google.com.br/books?id=HXC7jgEACAAJ.

[14] Ananth Grama, George Karypis, Vipin Kumar (Autor), and Anshul Gupta. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[15] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.