Test Suite Parallelization in Open-Source Projects: A Study on Its Usage and Impact

Jeanderson Candido, Luis Melo, and Marcelo d'Amorim

Abstract—Dealing with high testing costs remains an important problem in Software Engineering. Test suite parallelization is an important approach to address this problem. This paper reports our findings on the usage and impact of test suite parallelization in open-source projects. It provides recommendations to practitioners and tool developers to speed up test execution.
Considering a set of **[[468]]** popular Java projects we analyzed, we found that 24% of the projects contain costly test suites but parallelization features still seem underutilized in practice — only 19.1% of costly projects use parallelization. The main reported reason for adoption resistance was the concern to deal with concurrency issues. Results suggest that, on average, developers prefer high predictability than high performance in running tests.

1 Introduction

Dealing with high testing costs has been an important problem in software engineering research and industrial practice. Several approaches have been proposed in the research literature to address this problem, with the focus mainly on test suite minimization, prioritization, reduction, and selection [1]. In industry, the focus has been mainly on distributing the testing workload. Evidence of this are the Google TAP system [2], [3] and the Microsoft CloudBuild system [4], which provide distributed infrastructures to efficiently build massive amounts of code and run tests. Building in-house server clusters is also a popular mechanism to distribute testing workloads. For example, as of August 2013, the test suite of the Groupon PWA system, which powers the groupon.com website, included over 19K tests. To run all those tests under 10m, Groupon used a cluster of 4 computers with 24 cores each [5].

At large organizations, the alternative of renting cloud services [6] or even building proprietary infrastructures for running tests is a legitimate approach to mitigate the regression testing problem. However, for projects with modest or nonexistent budgets and yet relatively heavy testing workloads, this solution may not be economically viable. For these cases, the use of commodity hardware is an attractive solution for running tests. The proliferation of multi-core CPUs and the increasing popularization of testing frameworks and build systems, which today provide mature support for parallelization, enable speedups through increased CPU usage (see Section 2). These two elements — demand for cost-effective test execution and supply of relatively inexpensive testing infrastructures — inspired us to investigate test suite parallelization in practice.

This paper reports on an empirical study we conducted to analyze the usage and impact of low-level parallelization to speed up testing in open-source projects. This is a relevant problem given the tremendous popularity of open-source development and regression testing research [1]. Note that parallelization is complementary to other approaches to mitigate testing costs such as (safe) test selection [7], [8] and continuous integration [9].

The dimensions of analysis we considered in this study are (i) relevance, (ii) adoption, (iii) speedup, and (iv) tradeoffs. The dimension *relevance* focuses on the potential of parallelization to reduce testing costs. In the limit, parallelization would be fruitless if all projects had short-running test suites, for example. The dimension *adoption* evaluates how often existing open-source projects use parallelization schemes and how developers involved in costly projects (not using test suite parallelization) perceive this technology. It is important to measure resistance of practitioners to the technology and to understand their reasons. The dimension *speedup* evaluates the observed impact of parallelization in running times. Finally, the dimension *tradeoffs* evaluates the relationship between speedups obtained with parallelization and issues that arise when running tests in parallel, including test flakiness [10], [11]. We briefly summarize our findings in the following.

Relevance. To assess how prevalent long-running test suites are we selected [[468]] popular Java projects from Github containing Maven build files [12]. Section 4 details our methodology to select subjects and to isolate our experiments from environmental noise. Results indicate that nearly 24% of the projects take at least 1m to run and 8% of the projects take at least 5m to run. Considering the 110 projects with test suites taking longer than a minute to run, the average execution time of a test suite was 9m. Results also show that test cases are typically short-running, typically taking less than half a second to run. Furthermore, we found that only in rare cases few test cases monopolize the overall time to run a test suite.

Adoption. We considered two aspects in measuring technology adoption. First, we measured usage of parallelism in open-source projects. Then, we ran a survey with developers to understand the reasons that could explain resistance to using the technology. Considering only the projects whose test suites take longer than a minute to run, we found that only 19.1% of them use parallelism. We also contacted developers from a selection of costly projects that did not use

• *J. Candido, L. Melo, and M. d'Amorim are affiliated to the Department of Computer Science, Federal University of Pernambuco, Recife, PE, 50740-560 Brazil.*
E-mail: {jbc5, lhsm, damorim}@cin.ufpe.br

parallelization to understand the reasons for not using parallelization. Dealing with concurrency-related issues (e.g., the extra work to organize test suite to avoid concurrency errors) and the availability of continuous integration services were the most frequently answered reasons for not considering parallelization.
Speedups. We used two setups to measure speedup. In one setup we selected projects that run test suites in parallel by default, used a machine with eight virtual CPUs, and compared execution times of test suites with and withouth (default) parallelism enabled. In the other setup, we evaluated how execution scales with the number of available cores in the machine. For that, we used a more powerful machine with 80 virtual CPUs (40 cores with two native threads on each core). Considering the first setup, results indicate that the average speedup of parallelization was 3.53x. Although we found cases with very high speedups (e.g., 28.8x for project Jcabi), we also found cases where the speedups were not very significant. Considering the scalability experiment, we noticed, perhaps as expected, that parallelization obtained with forking JVMs scales with the number of cores but the speedup is bounded by long-running test classes.
Tradeoffs. Test flakiness is a central concern when running tests in parallel. Dependent tests can be affected by different schedulings of test methods and classes. This dimension of the study measures the impact of different parallel configurations on test flakiness and speedup. Overall, results indicate that configurations that fork JVMs do not achieve speedups as high as other more-aggressive configurations, but they manifest much lower flakiness ratios.

Our observations may trigger different actions:

- *Incentivize forking.* Forked JVMs manifest very low rates of test flakiness. Developers of projects with long-running test suites should consider using that feature, which is available in modern build systems today (e.g., Maven).
- *Break test dependencies.* Non-forked JVMs can achieve impressive speedups at the expense of sometimes impressive rates of flakiness. Breaking test dependencies (with ElectricTest [11], for example) to avoid flakiness is advised for developers with greater interest in efficiency.
- *Refactor tests for load balancing.* The configuration with forked JVMs scales better when the test workload is balanced across testing classes. Automated refactoring could help balance the workload in scenarios where developers are not willing to change test code but have access to machines with a high number of cores.
- *Improve debugging for build systems.* While preparing our experiments, we found scenarios where Maven's executions did not reflect corresponding JUnit's executions. Those issues can hinder developers from using parallel testing. Better debugging support for build systems could help on that.

The artifacts we produced as result of this study are available from the following web page https://jeandersonbc.github.io/testsuite-parallelization/.

2 Parallel Execution of Test Suites

Figure 1 illustrates different levels where parallelism in test execution can be obtained. The highest level indicates parallelism obtained through different machines on the network. For instance, using virtual machines from a cloud service to distribute test execution. The lowest levels denote parallelism obtained within a single machine. These levels are complementary: the lowest levels leverage the computing power of server nodes whereas the highest level leverages the aggregate processing power of a network of machines. This paper focuses on low-level parallelism, where computation can be offloaded at different CPUs within a machine and at different threads within each CPU. This form of parallelism is enabled through build systems (spawning processes in different CPUs) and testing frameworks (spawning threads in one given CPU). It is important to note that a variety of testing frameworks provide today support for parallel test execution (e.g., JUnit [13], TestNG [14], and NUnit [15]) as to benefit from the available power of popular multi-core processors. In the following, we elaborate relevant features of testing frameworks and build systems for parallelization. We focused on Java, Maven, and JUnit but the discussion can be generalized to other language and tools.

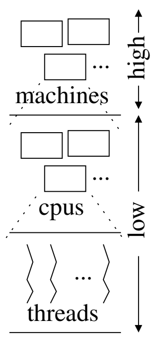

Fig. 1. Levels of parallelism.

2.1 Testing Frameworks

The list below shows the choices to control parallelism within one Java Virtual Machine (JVM). These options are offered by the testing framework (e.g., JUnit).

- **Sequential (C0).** No parallelism is involved.
- **Sequential classes; parallel methods (C1).** This configuration corresponds to running test classes sequentially, but running test methods from those classes concurrently.
- **Parallel classes; sequential methods (C2).** This configuration corresponds to running test classes concurrently, but running test methods sequentially.
- **Parallel classes; Parallel methods (C3).** This configuration runs test classes and methods concurrently.

Notice that an important aspect in deciding which configuration to use (or in designing new test suites) is the possibility of race conditions on shared data during execution. Data sharing can occur, for example, through state that is reachable from statically-declared variables in the program or through variables declared within the scope of the test class or even through resources available on the file system and the network [10]. Considering data race avoidance, configuration C1 is preferable over C2 when it is clear that test methods in a class do not manipulate shared state, which can be challenging to determine [11]. Similarly, C2 is preferable over C1 when it is clear that several test methods in a class perform operations involving shared data. Configuration C3 does not restrict scheduling orderings. Consequently, it is more likely to manifest data races during execution. Note that speedups depend on several factors, including the test suite size and distribution of test methods per class.

2.2 Build Systems

Forking OS processes to run test jobs is the basic mechanism of build systems to obtain parallelism at the machine space (see Figure 1). For Java-based build systems, such as Maven and Ant, this amounts to spawning one JVM, on a given CPU, to handle a test job and aggregating results when jobs finish. The list below shows the choices to control parallelism through the build system (e.g., Maven).

- **Forked JVMs with sequential methods (FC0).** The build system spawns multiple JVMs with this configuration, assigning a partition of the set of test classes to each JVM. Test classes and methods run sequentially within each JVM.
- **Forked JVMs with parallel methods (FC1).** With this configuration, the build system forks multiple JVMs, as FC0 does, but it runs test methods concurrently, as C1 does.

Note from the listing that forking can only be combined with configuration C1 (see Section 2.1) as Maven made the design choice to only accept one test class at a time per forked process. Maven offers an option to reuse JVMs that can be used to attenuate the potentially high cost of spawning new JVM processes on every test class (if reuse is enabled) and also to achieve test isolation (if reuse is disabled).

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
        <forkCount>1C</forkCount>
        <reuseForks>true</reuseForks>
        <parallel>methods</parallel>
        <threadCount>5</threadCount>
    </configuration>
</plugin>
```

Fig. 2. Configuration FC1 on Maven.

Example

Figure 2 shows a fragment of a Maven configuration file, known as *pom.xml*, highlighting options to run tests using the parallel execution mode FC1. Maven implements this feature through its Surefire JUnit test plugin [16]. With this configuration, Maven forks one JVM per core (`forkCount` parameter) and uses five threads (`threadCount` parameter) to run test methods (`parallel` parameter) within each forked JVM. Maven reuses created JVMs on subsequent forks when execution of a test class terminates (`reuseFork` parameter).

3 Objects of Analysis

We used Github's search API [17] to identify projects that satisfy the following criteria: (1) the primary language is Java[1], (2) the project has at least 100 stars, and (3) the latest update was on or after [[**Jean:** January 1st, 2017]]. Although there is no clearcut limit on the number of Github stars [18] to define relevant projects, we observed that one hundred stars was enough to eliminate trivial subjects. The third criteria serves to skip projects without recent activity. [[**Jean:** We focused on Java for its popularity. It is important to highlight that the Github's search API imposes a limit of returning only the top 1000 results for a single query. To overcome this limitation and have a larger sample set, we divided our original query into multiple queries over subsequent periods of time and considered the union the results.]] Figure 3 illustrates a query to the Github API as an HTTP request.

1. In case of projects in multiple languages, the Github API considers the predominant language as the primary language.

```
https://api.github.com/search/repositories?q=language:java
 +stars:>=100+pushed:2017-01-01..2017-02-01&page=1
```

Fig. 3. Query to the Github API for projects that (1) use Java, (2) contains at least 100 stars, (3) has been updated between January 1st and February 1st of 2017.

We used the following methodology to select projects for analysis. After obtaining the list of potential projects from GitHub, we filtered those with Maven support (i.e., contains a *pom.xml* file in the root directory). We focused on Maven for its popularity in Java projects. Then, considering this set of Maven projects, we executed the tests for three times to discard those projects with issues in the build file and non-deterministic results observed from sequential executions.

As of December 12th 2017, our search criteria returned a total of 7523 projects, and we identified 2094 Maven projects. [[From this set of projects, [[237]] projects were not considered because of environment incompatibility (e.g., missing DBMS) and [[13]] projects were discarded because of "flaky tests" [10]. A "flaky" test is a test that passes or fails under the same circumstances leading to non-deterministic results. As some of our experiments consist of running tests on different threads, we ignored these projects as it would be impractical to identify whether a test failed due to a race condition or some other source of flakiness. From the remaining [[533]] projects with deterministic results, we eliminated [[65]] projects with [[10%]] or more failing tests as to reduce bias. For the remaining projects with failing tests, we used the JUnit's `@Ignore` annotation to ignore failing tests. Our final set of subjects contains [[468]] projects. Figure 4 summarizes our sample set.]]

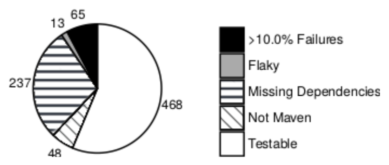

Fig. 4. We fetched 7523 popular projects hosted on Github. From this initial sample, we ignored [[48]] projects without Maven support, [[237]] with missing dependencies, [[13]] projects with flaky tests, and [[65]] projects had at least [[10%]] of failing tests. We considered [[468]] projects to conduct our study.

To run our experiments, we used a Core i7-4790 (3.60 GHz) Intel processor machine with eight virtual CPUs (four cores with two native threads each) and 16GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). We used Java 8 and Maven 3.5.2 to build projects and run test suites. To process test results and generate plots we used Python, Bash, R and Ruby. All source artifacts are publicly available for replication on our website [19]. This includes supporting scripts and the full list of projects.

4 Evaluation

We pose the following research questions, organized by the dimensions of analysis we presented in Section 1.

- Relevance
  - **RQ1.** How prevalent are time-consuming test suites?
  - **RQ2.** How is time distributed across test cases?
- Adoption
  - **RQ3.** How popular is test suite parallelization?
  - **RQ4.** What are the main reasons that prevent developers from using test suite parallelization?
- Speedups
  - **RQ5.** What are the speedups obtained with parallelization (in projects that actually use it)?
  - **RQ6.** How test execution scales with the number of available CPUs?
- Tradeoffs
  - **RQ7.** How parallel execution configurations affect testing costs and flakiness?

4.1 Relevance

- RQ1. **How prevalent are time-consuming test suites?**

To evaluate prevalence of projects with time-consuming test suites, we considered the [[468]] projects, appearing in Figure 4. Figure 5 illustrates the script we used to measure time.

We took the following actions to isolate our environment from measurement noise. First, we observed that some test tasks called test-unrelated tasks (e.g., *javadoc* generation and static analyses) that could interfere in our time measurements. To address that potential issue, we inspected Maven execution logs from a sample including a hundred projects prior to running the script from Figure 5. The tasks we found were ignored from execution (lines 1-4). Furthermore, we configured our workstation to only run essential services as to avoid noise from unrelated OS events. The machine was dedicated to our experiments and we accessed it via SSH. In addition, we configured the `isolcpus` option from the Linux Kernel [20] to isolate six virtual CPUs to run our experiments, leaving the remaining CPUs to run OS processes [21]. The rationale for this decision is to prevent context-switching between user processes (running the experiment) and OS-related processes. Finally, to make sure our measurements were fair, we compared timings corresponding to the sequential execution of tests using Maven with that obtained with JUnit's default `JUnitCore` runner, invoked from the command line. Results were very close. The main loop (lines 6-15) of the script in Figure 5 iterates over the list of subjects and invokes Maven multiple times (lines 8-11). It first makes all dependencies available locally (line 8), compiles the source and test files (line 9), and then runs the tests in offline mode as to skip the package update task, enabled by default (line 11). After that, we used a regular expression on the output log to find elapsed times (line 12-14).

We ran the test suite for each subject three times, reporting averaged execution times in three ranges: tests that run within a minute (short), tests that run in one to five minutes (medium), and tests that run in five or more minutes (long).

```
 1  MAVEN_SKIPS="-Drat.skip=true -Dmaven.javadoc.skip=true \
 2      -Djacoco.skip=true -Dcheckstyle.skip=true \
 3      -Dfindbugs.skip=true -Dcobertura.skip=true \
 4      -Dpmd.skip=true -Dcpd.skip=true"
 5
 6  for subj in $SUBJECTS; do
 7      cd $SUBJECTS_HOME/$subj
 8      mvn clean dependency:go-offline
 9      mvn test-compile install -DskipTests $MAVEN_SKIPS \
10          &> compile.log
11      mvn test -o -fae $MAVEN_SKIP &> testing.log
12      cat testing.log \
13          | grep --text "\[INFO\] Total time:" \
14          | tail -n 1
15  done
```

Fig. 5. Bash script to measure time cost of test suites. For each subject, we fetch all dependencies, compile the source and test files, and execute the tests in offline mode ignoring non-related tasks. Test-unrelated tasks are omitted.

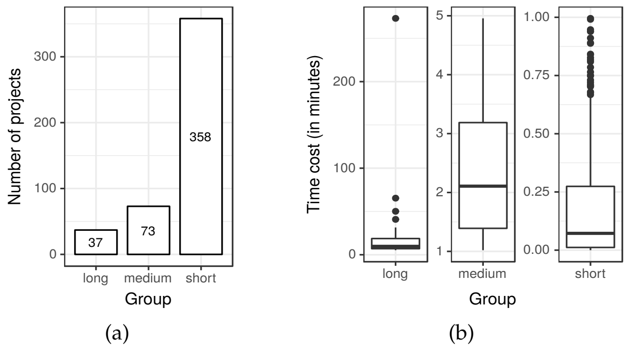

Fig. 6. (a) Number of projects in each cost group and (b) Distribution of running times per cost group.

Figure 6(a) shows the number of projects in each group. As expected, long and medium projects do not occur as frequently as short projects. However, they do occur in relatively high numbers. Figure 6(b) shows the distribution of execution time of test suites in each of these groups. Note that the y-ranges are different. The distribution associated with the short group is the most unbalanced (right skewed). The test suites in this group ran in 15 or less seconds for over 75% of the cases. Considering the groups medium and long, however, we found many costly executions. Nearly 75% of the projects from the medium group take 3.5 or more minutes to run and nearly 75% of the projects from the long group take ~20 minutes to run. We found cases in the long group were execution takes more than 50 minutes to complete, as can be observed from the outliers in the boxplot.

It is important to note that we under-estimated running times as we missed test modules not enabled for execution in the root *pom.xml*. For instance, the project `apache.maven-surefire` runs all unit tests in a few seconds. According to our criteria, this project is classified as short but a closer look reveals that only smoke tests are executed in this project by default. In this project, integration and system tests, which take longer to run, are only accessible via custom parameters, which we do not handle in our experimental setup. We enabled such parameters for this specific project and observed that testing time goes to nearly 30 minutes. For simplicity, we considered only the tests executed by default. From the [[468]] testable projects, 400 successfully executed all tests and 68 reported some test failure. From these 68 subjects, only 11 subjects have more than 5% of failing tests (7.3% on average).

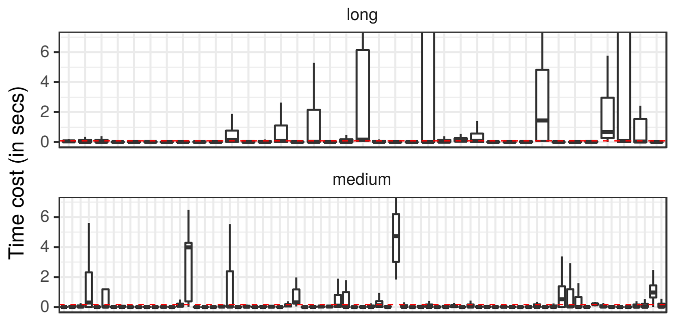

Fig. 7. Distribution of test case time per project.

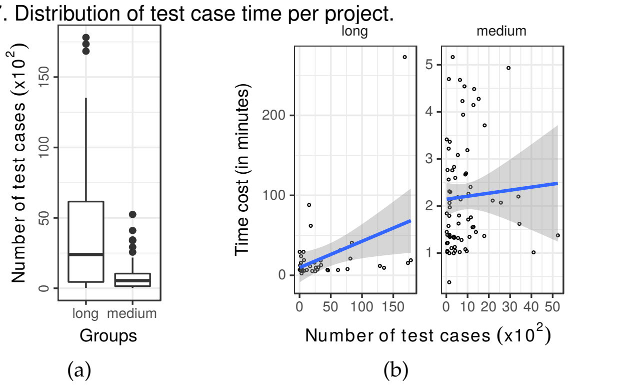

Fig. 8. (a) Size of test suites; (b) Size versus running time of test suites.

Answering RQ1: We conclude that time-consuming test suites are relatively frequent in open-source projects. We found that 24% of the [[468]] projects we analyzed (i.e., nearly 1 in every 4 projects) take at least 1 minute to run and 8% of them take at least 5 minutes to run.

- RQ2. **How is time distributed across test cases?**

Section 4.1 showed that medium and long-running projects are not uncommon, accounting to nearly 24% of the [[468]] projects we analyzed. Research question RQ2 measures the distribution of test costs in test suites. In the limit, if cost is dominated by a single test from a large test suite, it is unlikely that parallelization will be beneficial as a test method is the smallest working unit in test frameworks. Figure 7 shows the time distribution of individual test cases per project. We observed that the average median times (see dashed horizontal red lines) were small, namely 0.08s for medium projects and 0.16s for long projects, and the standard deviations associated with each distribution were relatively low. High values of σ are indicative of CPU monopolization. We found only a small number of those. The highest value of σ occurred in `uber_chaperone`, a project from the long group. This project contains only 26 tests, 17 of which take less than 0.5s to run, one of which takes nearly 3s to run, two of which take nearly 11s to run, four of which takes on average 3m to run, and two of which take $\sim$8m to run. For this project, 98.4% of the execution cost is dominated by 20% of the tests; without these two costly tests this project would have been classified as short-running. We did not find other projects with such extreme time monopolization profile. Project `facebookarchive_linkbench` is also classified as long-running and has the second highest value of σ. For this project, however, cost is distributed more smoothly across 98 tests, of which 8 (8.1%) take more than 1s to run with the rest of the tests running faster.

Figure 8(a) shows the difference in the distribution of test suite sizes across groups. This figure indicates that long projects have a higher median and much higher average number of test cases. Furthermore, we noted a strong positive correlation between running time and number of test on projects in the long group. Considering the medium group, the correlation between these two variables was weak. Figure 8(b) illustrates the regression lines between these the variables test suite cost and number of test cases. To sum, we observed that for projects with long-running test suites running time is typically justified by the number of test cases as opposed to the cost of individual test cases.

Answering RQ2: Overall, results indicate that projects with a very small number of tests monopolizing end-to-end execution time were rare. Time most often is distributed evenly across test cases.

4.2 Adoption

- RQ3. **How popular is test suite parallelization?**

To answer RQ3 we used projects from the medium and long groups where parallelization can be more helpful. We used a dynamic and a static approach to find manifestations of parallelism. We discuss results obtained with these complementary approaches in the following.

4.2.1 *Dynamic checking*

To find dynamic evidence of parallelism, we ran the test suites from our set of 110 projects to output all key-value pairs of Maven parameters. To that end, we used the option `-X` to produce debug output and the option `-DskipTests` to skip execution of tests. We skipped execution of tests as we observed from sampling that only bootstrapping the Maven process suffices to infer which parallel configuration modes it uses to run the tests. It is also important to point that we used the default configurations specified in the project. We inferred parallel configurations by searching for certain configuration parameters in log files. According to Maven's documentation [16], a parallel configuration depends either on (1) the parameter `parallel` to define the parallelism mode within a JVM followed by the parameter `threadCount` or (2) the parameter `forkCount`[2] to define the number of forked JVMs. As such, we captured, for each project, all related key-value pairs of Maven parameters and mapped those pairs to one of the possible parallelization modes. For instance, if a given project contains a module with the parameter `<forkCount>1C</forkCount>`, the possible classifications are FC0 or FC1, depending on the presence and the value of the parameter `parallel`. If the parameter `parallel` is set to `methods` the detected mode will be FC1. Large projects may contain several test suites distributed on different Maven modules potentially using different configurations. For those cases, we collected the Maven output from each module discarding duplicates as to avoid inflating counts for configuration modes that appear in several modules of the same project. For instance, if a project contains two modules using the same configuration, we counted only one occurrence. Considering our set of 110 projects, we found that only 13 of those projects had

2. This parameter is named `forkMode` in old versions of Maven Surefire.

parallelism enabled by default, with only configurations C2, C3, and FC0 being used. Configurations C3 and FC0 were the most popular among these cases. Note that these results under-approximate real usage of parallelism as we used default parameters in our scripts to spawn the Maven process. That decision could prevent execution of particular test modules. Table 1 shows the 13 projects we identified where parallelism is enabled by default in Maven.

Column *"Subject"* indicates the name of the project, column *"# of modules"* indicates the fraction of modules containing tests that use the configuration of parallelism mentioned in column *"Mode"*. We note that, considering these projects, the modules that do not use the configuration cited use the sequential configuration C0. For example, three modules (=28-25) from Log4J2 use sequential configuration. It came as a surprise the observation that no project used distinct configurations in their modules.

TABLE 1
Subjects with parallel test execution enabled by default.

Group	*Subject*	*# of modules*	*Mode*
Medium	Californium	2/20	C2
Medium	Chaos	1/1	C2
Long	Flink	66/74	FC0
Long	Log4J2	25/28	FC0
Long	Javaslang	3/3	C3
Medium	Jcabi	1/1	C3
Long	Jet	6/7	FC0
Long	Mahout	8/9	FC0
Long	MapDB	1/1	C3
Medium	OpenNLP	4/4	FC0
Medium	Rultor	1/1	C3
Medium	Takes	1/1	C3
Long	Vavr	3/3	C3

4.2.2 Static checking

Given that the dynamic approach cannot detect parallelism manifested through the default configuration of projects, we also searched for indications of parallelism in build files. We parsed all *pom.xml* files under the project's directory and used the same approach as in our previous analysis to classify configurations. We noticed initially that our approach was unable to infer the configuration mode for cases where the decision depends on the input (e.g., `<parallel>${parallel.type}</parallel>`). For these projects, the tester needs to provide additional parameters in the command line to enable parallelization (e.g., `mvn test -Dparallel.type=classesAndMethods`). To handle those cases, we considered all possible values for the parameter (in this case, `${parallel.type}`). It is also important to note that this approach is not immune to false negatives, which can occur when *pom.xml* files are encapsulated in jar files or files downloaded from the network. Consequently, this approach complements the the dynamic approach. Overall, we found 14 projects manifesting parallelism with this approach. Compared to the set of projects listed in Table 1, we found four new projects, namely: `Google Cloud DataflowJavaSDK` (using configuration C3), `Mapstruct` (using configuration FC0), `T-SNE-Java` (using configuration FC0), and `Urbanairship Datacube` (using configuration C3). Curiously, we also found that project `Jcabi`, `Rultor`, and `Takes` were not detected using this methodology. That happened because these projects loaded a *pom.xml* file from a jar file that we missed. Considering the static and dynamic methods together, we found a total of 17 distinct projects using parallelism, corresponding to the union of the two subject sets.

Answering RQ3: Results indicate that test suite parallelization is underused. Overall, only 15.45% of costly projects (17 out of 110) use parallelism.

- RQ4. **What are the main reasons that prevent developers from using test suite parallelization?**

To answer this research question we surveyed developers involved in a selection of projects from our benchmark with time-consuming test suites. The goal of the survey is to better comprehend developer's attitude towards the use of parallelism as a mechanism to speedup regression testing. We surveyed developers from a total of 89 projects. From the initial list of 110 project, we discarded 11 projects that we knew a priori used parallelization, and 10 projects that we could not find developer's emails from commit logs. From this list of projects, we mined potential participants for our study. More precisely, we searched for developer's name and email from the last 20 commits to the corresponding project repository. Using this approach, we identified a total of 297 eligible participants. Finally, we sent plain-text e-mails, containing the survey, to those developers. In total, 38 developers replied but we discarded three replies with subjective answers. Considering projects covered by the answers, a total of 36 projects (61.29% of the total) were represented in those replies. Note that multiple developers on each project received emails. In one specific case, one developer worked in multiple projects, and we consider it as a different answer. We sent the following set of questions to developers:

1) How long does it take for tests to run in your environment? Can you briefly define your setup?
2) Do you confirm that your regression test suite does *not* run in parallel?
3) Select a reason for not using parallelization:
   a) I did not know it was possible
   b) I was concerned with concurrency issues
   c) I use a continuous integration server
   d) Some other reason. Please elaborate.

Considering question 1, we confirmed that execution time was compatible with the results we reported in Section 4.1. Furthermore, 12 of the participants indicated the use of Continuous Integration (CI) to run tests, with 4 of these participants reporting that test suites are modularized and those modules are tested independently in CI servers through different parameters. Those participants explained that such practice helps to reduce time to observe test failures, which is the goal of speeding up regression testing. A total of 6 participants answered that they do run tests in their local machines. Note, however, that CI does not preclude low-level parallelization. For example, installations of open-source CI tools (e.g., Jenkins [22]) in dedicated servers would benefit from running tests faster through low-level test suite parallelization.

Considering question 2, the answers we collected indicated, to our surprise, that six of the 36 projects execute tests in parallel. This mismatch is justified by cases where neither of our checks (static or dynamic) could detect presence of parallelism. A closer look at these projects revealed that one of them contained a *pom.xml* file encapsulated in a jar file (similar case as reported in Section 4.2.2), in one of the projects the participant considered that distributed CI was a form of parallelism, and in four projects the team preferred to implement parallelization instead of using existing features from the testing framework and the build system — in two projects the team implemented concurrency control with custom JUnit test runners and in two other projects the team implemented concurrency within test methods. Note that, considering these four extra cases (ignored two distributed CI cases), the usage of parallelization increases from 15.45% to 19.1%. We do not consider this change significant enough to modify our conclusion about practical adoption of parallelization (RQ3).

Considering question 3, the distribution of answers was as follows. A total of 8.33% of the 36 developers who answered the survey did not know that parallelism was available in Maven (option "a"), 33.33% of developers mentioned that they did not use parallelism concerned with possible concurrency issues (option "b"), 16.67% of developers mentioned that continuous integration suffices to provide timely feedback while running only smoke tests (i.e., short-running tests) locally (option "c"), and 16.67% of developers who provided an alternative answer (option "d") mentioned that using parallelism was not worth the effort of preparing the test suites to take advantage of available processing power. A total of 19.45% of participants did not answer the last question of the survey. The pie chart in Figure 9 summarizes the distribution of answers.

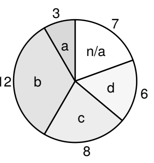

Fig. 9. Summary of developer's answers to survey question 3.

Answering RQ4: Results suggest that dealing with concurrency issues (i.e., the extra work to organize test suite to safely explore concurrency) was the principal reason for developers not investing in parallelism. Other reasons included availability of continuous integration services and unfamiliarity with the technology.

4.3 Speedups

- RQ5. **What are the speedups obtained with parallelization (in projects that actually use it)?**

To answer RQ5, we considered the 13 subjects from our benchmark that use parallelization *by default* (see Table 1). We compared running times of test suites with enabled parallelization, as configured by project developers, and without parallelization. It is important to note that there are no observed failures in either execution. Table 2 summarizes results. Lines are sorted by project names. Columns "*Group*" and "*Subject*" indicate, respectively, the cost group and the name of the project. Column "T_s" shows sequential execution time and column "T_p" shows parallel execution time. Column "T_s/T_p" shows speedup or slowdown. As usual, a ratio above 1x indicates speedup and a ratio below 1x indicates slowdown.

Results show that, on average, parallel execution was 3.53 times faster compared to sequential execution. Three cases worth special attention: `Log4J2`, `Chaos`, and `Takes`.

TABLE 2
Speedup (or slowdown) of parallel execution (T_p) over sequential execution (T_s). Default parallel configuration of Maven is used. Highest slowdown/speedup appears in gray color.

Group	*Subject*	T_s	T_p	T_s/T_p
Medium	Californium	1.45m	1.40m	1.04x
Medium	Chaos	1.51m	1.47m	1.03x
Medium	Flink	11.79m	2.57m	4.59x
Long	Log4J2	8.24m	8.21m	1.00x
Medium	Javaslang	2.18m	1.82m	1.20x
Medium	Jcabi	2.76m	0.30m	9.20x
Long	Jet	8.26m	3.67m	2.25x
Long	Mahout	27.38m	18.15m	1.51x
Long	MapDB	10.06m	8.58m	1.17x
Medium	OpenNLP	1.30m	0.55m	2.36x
Medium	Rultor	2.30m	0.27m	8.52x
Medium	Takes	2.00m	0.19m	10.53x
Long	Vavr	3.26m	2.25m	1.45x
Average				3.53x

We note that parallel execution in `Log4J2` was ineffective. We found that Maven invokes several test modules in this project but the test modules that dominate execution time run sequentially by default. This was also the case for the highlighted project `Californium`. No significant speedup was observed in `Chaos`, a project with only three test classes, of which one monopolizes the bulk of test execution time. This project uses configuration C2, which runs test classes in parallel but runs test methods, declared in each class, sequentially. Consequently, speedup cannot be obtained as the cost of the single expensive test class cannot be broken down with the selected configuration. Finally, the speedup observed in project `Takes` was the highest amongst all projects. This subject uses configuration C3 and contains 419 test methods distributed nearly equally among 148 test classes with a small number of test methods. Furthermore, several methods in those classes are time-consuming. As result, the CPUs available for testing are kept occupied for the most part during test execution.

Answering RQ5: Considering the machine setup we used, the average speedup observed with default configurations of parallelization was 3.53x.

- RQ6. **How test execution scales with the number of available CPUs?**

This experiment evaluates the impact of making a growing number of CPUs available to the build system for testing. For this reason, we used a different machine, with more cores, compared to the one described in Section 3. We used a Xeon E5-2660v2 (2.20GHz) Intel processor machine with 80 virtual CPUs (40 cores with two native threads each) and 256GB of memory, running Ubuntu 14.04 LTS Trusty Tahr (64-bit version). This experiment spawns a growing number of JVMs in different CPUs, using parallel configuration *FC0*. We selected subject MapDB in this experiment as it represents the case of a long-running test suite (see Table 2) with test cases distributed across many test classes – 194. Recall that a test class is the smallest unit

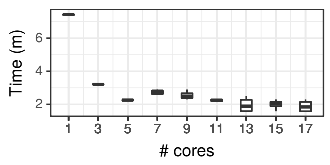

Fig. 10. Scalability.

that can be used to spawn a test job on a JVM and that we have no control over which test classes will be assigned to which JVM that the build system forks. Figure 10 shows the reduction in running times as more CPUs contribute to the execution. We ran this experiment for a growing number of cores 1, 3, ..., 39. The plot omits results beyond 17 cores as the tendency for higher values is clear. We noticed that improvements are marginal after three cores, which is the basic setup we used in other experiments. This saturation is justified by the presence of a single test class, `org.mapdb.WALTruncat`, containing 15 test cases that take over two minutes to run.

Answering RQ6: Results suggest that execution FC0 scales with additional cores but there is a bound on the speedup that one can get related to how well the test suite is balanced across test classes.

4.4 Tradeoffs

This dimension assesses the impact of using distinct parallel configurations on test flakiness and speedup. Increased parallelism can increase resource contention leading to concurrency issues such as data races across dependent tests [10], [11]. Flakiness and speedup are contradictory forces that could influence the decision of practitioners about which parallel configuration should be used for testing. Note that Section 4.3 evaluated speedup in isolation.

- RQ7. **How parallel execution configurations affect testing costs and flakiness?**

To answer this research question, we selected 15 different subjects, ran their test suites against all configurations described in Section 2, and compared their running times and rate of test flakiness. We used the sequential execution configuration, *C0*, as the comparison baseline in this experiment. To select subjects, we sorted projects whose test suites run in 1m or more by decreasing order of execution time and selected the first fifteen projects that use JUnit 4.7 or later. The rationale for this criteria is to ensure compatibility with parallel configuration since older versions of JUnit does not support parallel testing. We ran each project on each configuration for three times. Overall, we needed to reran test suites 270 times, 18 times (3x6 configurations) on each project. Given the low standard deviations observed in our measurements, we considered three reruns reasonable for this experiment.

It is worth mentioning that we used custom JUnit runners as opposed to Maven to run the test suites with different parallel configurations (see Section 2). After carefully checking library versions for compatibility issues and comparing results with JUnit's we observed that several of Maven's executions exposed problems. For example, Maven incorrectly counts the number of test cases executed for some of the cases where test flakiness are observed. These issues are categorized and documented on our website [19] and can be reproduced with our scripts. To address those issues we implemented custom test runners for configurations *C1*, *C2*, and *C3* and, for configurations *FC0* and *FC1*, we implemented a bash script that coordinates the creation of JVMs and invokes corresponding custom runners. As to faithfully reflect Maven's behavior in our scripts, we carefully analyzed the source code [23] of the Maven Surefire plugin. We implemented test runners using the `ParallelComputer` class from JUnit [24].

We used Maven log files to identify test classes to run and used the Maven dependency plugin [25] to build the project's classpath (with the command `mvn dependency:build-classpath`). Once we find the tests suite to run and the corresponding classpath, we invoke the test runners mentioned above on them. We configured this experiment to run at most three JVMs in parallel. Recall that in our setup (see Section 3), we limited our kernel to use only three cores and reserved one core for OS-related processes. To ensure that our experiments terminate (recall that deadlock or livelock could occur) we used the `timeout` command [26] configured to dispatch a *kill* signal if test execution exceeds a given time limit. Finally, we save each execution log and stack traces generated from JUnit to collect the execution time, the number of failing tests, and to diagnose outliers in our results.

Table 3 summarizes results ordered by subject's name. Values are averaged across multiple executions. We did not report standard deviations as they are very small in all cases. As to identify the potential causes of flakiness, we inspected the exceptions reported in execution logs. We found that, in most of the cases, flakiness was caused by race conditions: approximately 97.5% of the failures were caused by a `null` dereference and 1.6% were caused by concurrent access on unsynchronized data structures. Cases of likely broken test dependencies were not as prevalent as race conditions (0.8% of the total): `EOFException` (0.2%), `FileSystemAlreadyExistsException` (0.2%), and `BufferOverflowException` (0.4%). Results suggest that anticipating race conditions to schedule test executions would have higher impact compared to breaking test dependencies using a tool such as ElectricTest [11].

The projects with flakiness in all configurations were `AWS SDK`, `GoogleCloud`, and `Moquette`. It is worth highlighting the unfortunate case of `Moquette`, which manifested more than 20% flaky tests in every configuration. Considering time, it is noticeable from the averages, perhaps as expected, an increasing speedup from configuration *C1* to *C3* and from configuration *FC0* to *FC1*. It is also worth mentioning that some combinations manifested slowdown instead of speedup. Recall that parallel execution introduces the overhead of spawning and managing JVMs and threads. Overall, results show that 0% of flakiness have been reported in 30 of the 75 (=5x15) pairs of project and configuration we analyzed (40% of the total). In for 4 of the 15 projects flakiness was not manifested in any combination pairs. We noticed with some surprise that the average speedup of configuration *C1* was higher compared to *FC1* indicating that it is not always the case that using more CPUs pays off. Important to note that the cost of spawning new JVMs can be significant in *FC1*.

Answering RQ7: Overall results indicate that the test suites of 73.33% of the projects we analyzed could be run in parallel without manifesting any flaky tests. In some of these cases, speedups were significant, ranging from 1x to 28.8x.

TABLE 3
Speedup versus Flakiness (%$_{\text{fail}}$). Configuration *C0* denotes the comparison baseline. Columns T and N indicate time and number of tests, respectively. Other columns show speedup and percentage of failing tests in different configurations, compared to *C0*.

Subject	*C0*		*C1*		*C2*		*C3*		*FC0*		*FC1*	
	T	N	speedup	%$_{\text{fail}}$	speedup	%$_{\text{fail}}$	speedup	%$_{\text{fail}}$	speedup	%$_{\text{fail}}$	speedup	%$_{\text{fail}}$
Activiti	5.9m	2,029	72.1x	96.3%	1.5x	6.9%	75.9x	96.3%	2.9x	6.6%	3.1x	8.0%
AWS SDK Java Core	3.7m	847	2.0x	2.2%	2.5x	2.8%	3.7x	4.0%	1.9x	0.2%	3.5x	3.1%
Bucket4J	3.0m	110	2.5x	0%	1.3x	0.9%	4.2x	1.8%	1.3x	0%	3.7x	0%
Facebook Linkbench	6.1m	98	1.0x	0%	1.6x	1.0%	1.0x	0%	1.7x	0%	1.6x	0%
GoogleCloud Dataflow Java SDK	1.6m	3,345	1.2x	1.7%	2.7x	1.0%	0.8x	5.4%	0.8x	1.7%	0.8x	1.7%
INRIA spoon	2.3m	1,042	1.2x	28.8%	2.7x	77.2%	1.6x	56.6%	1.8x	0%	1.8x	29.0%
Jcabi Github	2.6m	634	2.1x	0%	17.7x	0%	28.8x	0%	2.0x	0%	2.9x	0%
JCTools Core	3.6m	690	4.5x	0%	3.6x	0%	18.0x	0%	2.8x	0%	9.0x	0%
MapDB	8.2m	5,324	1.5x	0%	2.7x	0%	4.8x	0%	1.7x	1.0%	3.4x	1.0%
Moquette	3.7m	169	4.6x	65.6%	3.4x	33.0%	12.3x	78.0%	2.5x	22.5%	9.3x	69.4%
Spring Cloud Function	2.8m	168	6.4x	77.4%	1.3x	0.6%	6.6x	79.2%	1.1x	0%	2.9x	32.7%
Stream Lib	2.1m	149	0.9x	0%	2.2x	0%	2.4x	0.7%	2.7x	0%	3.6x	0%
Stripe Java	4.3m	302	4.8x	6.3%	3.3x	7.3%	21.5x	15.0%	2.7x	0%	8.6x	11.6%
TabulaPDF Java	2.4m	186	7.2x	0.5%	1.1x	0%	7.2x	2.7%	1.0x	0%	7.2x	1.6%
Urban Airship Datacube	8.3m	36	1.9x	25.0%	4.7x	44.4%	1.9x	25.0%	1.0x	0%	1.9x	25.0%
Average	4.0m	1,006.6	7.6x	20.3%	3.5x	11.7%	12.7x	24.3%	1.9x	2.1%	4.2x	12.2%

5 Discussion

This paper reports our finding on a study to evaluate impact and usage of test suite parallelization, enabled by modern build systems and testing frameworks. This study is important given the importance to speedup testing. Note that test suite parallelization is complementary to alternative approach to speedup testing (see Section 7). The observations we made in this study trigger multiple actions:

- *Incentivize forking.* Forked JVMs manifest low rates of test flakiness. For instance, in *FC0*, only 4 of 10 projects manifest flakiness and, excluding the extreme case of `Moquette`, projects manifest flaky tests in low rates 0.23% to 1.70%. Developers of projects with long-running test suites should consider using that feature, which is available in modern build systems today (e.g., Maven).
- *Break test dependencies.* Non-forked JVMs can achieve impressive speedups at the expense of sometimes impressive rates of flakiness. Breaking test dependencies to avoid flakiness and take full advantage of those options is advised for developers with greater interest in efficiency.
- *Refactor tests for load balancing.* Forked JVMs scales better with the number of cores when the test workload is balanced across testing classes. To balance the workload, automated user-oblivious refactoring can help in scenarios where developers are not willing to change test code but have access to machines with a high number of cores.
- *Improve debugging for build systems.* While preparing our experiments, we found scenarios where Maven's executions did not reflect corresponding JUnit's executions. Those issues can hinder developers from using parallel testing. Better debugging infrastructure is important.

This study brings to light the benefits and burdens of test suite parallelization to improve test efficiency. It provides recommendations to practitioners and developers of new techniques and tools aiming to speed up test execution with parallelization.

6 Threats to Validity

The main threats to validity of this study are the following.

External Validity: Generalization of our findings is limited to our selection of subjects, testing framework, and build system. To mitigate that issue, we selected subjects according to an objective criteria, described in Section 3. It remains to evaluate the extent to which our observations would change when using different testing frameworks and build systems. Also, some of the selected subjects contain failing tests. Test failures may reduce the testing time due to early termination or even inflate the time (e.g., test waiting indefinitely for an unavailable resources). To mitigate this threat, we eliminated subjects with flaky tests and filtered projects with at least 90% of the tests passing. Only 17% of our subjects have failing tests. We carefully inspected our rawdata to identify and ignore these failures with JUnit's `@Ignore` annotation.

Internal Validity: Our results could be influenced by unintentional mistakes made by humans who interpreted survey data and implemented scripts and code to collect and analyze the data. For example, we developed JUnit runners to reproduce Maven's parallel configurations and implemented several scripts to automate our experiments (e.g., run tests and detect parallelism enabled by default in the subjects). All those tasks could bias our results. To mitigate those threats, the first two authors of this paper validated/inspected each other to increase chances of capturing unintentional mistakes.

Construct Validity: We considered a number of metrics in this study that could influence some of our interpretations. For example, we measured number of test cases per suite, distribution of test costs in a suite, time to run a suite, etc. In principle, these metrics may not reflect the main problems associated with test efficiency.

7 Related Work

Regression testing research has focused mostly on test suite minimization, prioritization, reduction, and selection [1], [27]. Most of these techniques are unsound (i.e., they do not guarantee that fault-revealing tests will be considered for testing). The test selection technique Ekstazi [8], [28] is an example of a sound regression testing technique. It conservatively computes which tests have been impacted by file changes. A test is discarded for execution if it does not depend on any changed file dynamically reachable from execution. Important to note that regression testing techniques, including test selection, is complementary to test suite parallelization.

ElectricTest [11] is a tool for efficiently detecting data dependencies across test cases. Dependency tracking is important as to avoid test flakiness when parallelizing test suites. ElectricTest observes reads and writes on global resources made by tests to identify these dependencies at low cost. We remain to investigate the impact of ElectricTest to reduce flakiness in unrestricted test suite parallelization.

The use of the Single Instruction Multiple Data (SIMD) design has been previously explored in research to accelerate test execution [29], [30], [31], [32], [33], [34], [35]. The SIMD architecture, as implemented in modern GPUs, for instance, allows the execution of a given instruction simultaneously against multiple data. For that reason, in principle, one test could be ran simultaneously against multiple inputs provided that multiple test inputs exist associated to that one test. Recent work [33], [35] explored that idea to speedup test execution of embedded software using graphic cards. Although benchmarks indicate superior performance compared to traditional multicore CPUs, the use of the technology in broader settings is limited. For example, execution of more general programs can violate the SIMD's lock-step assumption on the control-flow of threads. This violation would affect negatively performance. Furthermore, handling complex data is challenging in SIMD [29], [30]. The approach is promising when multiple input vectors exist for each test and the testing code heavily manipulates scalar data types. The datasets used in those papers satisfied those constraints.

Google [2], [3] and Microsoft [4] have been creating distributed infrastructures to efficiently build massive amounts of code and run massive amounts of tests. Those scenarios bring different and challenging problems such as deciding when to trigger the build under multiple file updates [36]. Although such distributed systems are targeted to extremely large scale code and test bases, the same ideas can be applied to handle the build process of large, albeit not as large, projects. For example, Gambi et al. [37] recently proposed CUT, a tool to automatically parallelize JUnit tests on the cloud. The tool allows the developer to control resource allocation and deal with the project specific test dependencies. Note that test suite parallelization is complementary to these high-level parallelism schemes.

Continuous Integration (CI) services, such as Travis CI [38], are becoming widely used in the open-source community [39], [40]. Accelerating time to run tests in CI is important as to reduce the period between test report updates. Module-level regression testing [41], for example, can be helpful in that setting. It is important to note that test failures are more common in CI compared to an overnight run or a local run, for instance. This can happen because of semantic merge conflicts [42], for instance. As such effect can impact developer's perception and tolerance towards failures, we are curious to know if developers would be willing to receive more frequent test reports at the expense of potentially increasing failure rates due to flakiness caused by parallelism.

8 Conclusions

Testing is expensive. Despite all advances in regression testing research, dealing with high testing costs remains an important problem in Software Engineering. This paper reports our findings on the usage and impact of test execution parallelization in open-source projects. Multicore CPUs are widely available today. Testing frameworks and build systems that capitalize on these machines also became popular. Despite some resistance observed from practitioners, our results suggest that parallelization can be used in many cases without sacrificing reliability. More research needs to be done to improve automation (e.g., breaking test dependencies, refactoring test suites, enforcing safe test schedules) as to safely optimize parallel execution. The artifacts we produced as result of this study are available from the following web page:

https://jeandersonbc.github.io/testsuite-parallelization/

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2012.

[2] G. E. Tools, "Testing at the speed and scale of google," June 2011, http://google-engtools.blogspot.com.br/2011/06/testing-at-speed-and-scale-of-google.html.

[3] GoogleTechTalks, "Tools for continuous integration at google," October 2010, http://www.youtube.com/watch?v=b52aXZ2yi08.

[4] C. Prasad and W. Schulte, "Taking control of your engineering tools," *Computer*, vol. 46, no. 11, pp. 63–66, 2013.

[5] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim, "SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems," in *ESEC/FSE*, 2013, pp. 257–267.

[6] "List of popular hosting cloud services," https://clutch.co/cloud.

[7] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, Apr. 1997. [Online]. Available: http://doi.acm.org/10.1145/248233.248262

[8] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *ISSTA*, 2015, pp. 211–222.

[9] D. Saff and M. D. Ernst, "Reducing wasted development time via continuous testing," in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ser. ISSRE '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 281–. [Online]. Available: http://dl.acm.org/citation.cfm?id=951952.952340

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 643–653. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635920

[11] J. Bell, G. Kaiser, E. Melski, and M. Dattatreya, "Efficient dependency detection for safe java test acceleration," in *ESEC/FSE*, 2015, pp. 770–781.

[12] Maven, 2017, https://maven.apache.org/.

[13] JUnit, "Junit web site," 2017, http://junit.org.

[14] TestNG, "Testng web site," 2017, http://testng.org.

[15] NUnit, "Nunit web site," 2017, http://www.nunit.org.

[16] M. S. Plugin, 2017, http://maven.apache.org/surefire/maven-surefire-plugin/.

[17] Github, "Github api web site," 2017, http://developer.github.com/v3/search/.

[18] ——, "Github about stars," 2017, https://help.github.com/articles/about-stars/.

[19] "Our web page," https://jeandersonbc.github.io/testsuite-parallelization/, 2017.

[20] Kernel.org, "Kernel linux options," https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html?highlight=isolcpu, 2017.

[21] U. S. community, "Using `isolcpus`," http://unix.stackexchange.com/questions/326579/how-to-ensure-exclusive-cpu-availability-for-a-running-process, 2017.

[22] "Jenkins ci," https://jenkins.io/, 2017.

[23] "Maven surefire source repository," http://svn.apache.org/viewvc/maven/surefire/trunk/, 2017.

[24] JUnit, "Parallelcomputer (junit api)," 2017, http://junit.org/junit4/javadoc/4.12/org/junit/experimental/ParallelComputer.html.

[25] M. D. Plugin, 2017, https://maven.apache.org/plugins/maven-dependency-plugin/.

[26] L. man Page, "timeout - run a command with a time limit," http://linux.die.net/man/1/timeout, 2017.

[27] Q. D. Soetens, S. Demeyer, A. Zaidman, and J. Pérez, "Change-based test selection: An empirical evaluation," *Empirical Softw. Engg.*, vol. 21, no. 5, pp. 1990–2032, Oct. 2016.

[28] A. Çelik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across JVM boundaries," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 809–820. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106297

[29] M. d'Amorim, S. Lauterburg, and D. Marinov, "Delta execution for efficient state-space exploration of object-oriented programs," in *ISSTA*, 2007, pp. 50–60.

[30] ——, "Delta execution for efficient state-space exploration of object-oriented programs," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 597–613, Sep. 2008.

[31] C. H. P. Kim, S. Khurshid, and D. Batory, "Shared execution for efficiently testing product lines," in *ISSRE*, 2012, pp. 221–230.

[32] H. V. Nguyen, C. Kästner, and T. N. Nguyen, "Exploring variability-aware execution for testing plugin-based web applications," in *ICSE*, 2014, pp. 907–918.

[33] A. Rajan, S. Sharma, P. Schrammel, and D. Kroening, "Accelerated test execution using gpus," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 97–102. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642957

[34] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *ESEC/FSE 2015*, 2015, pp. 842–853.

[35] V. Yaneva, A. Rajan, and C. Dubach, "Compiler-assisted test acceleration on gpus for embedded software," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 35–45. [Online]. Available: http://doi.acm.org/10.1145/3092703.3092720

[36] A. M. Memon, Z. Gao, B. N. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 233–242. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.16

[37] A. Gambi, S. Kappler, J. Lampel, and A. Zeller, "Cut: Automatic unit testing in the cloud," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 364–367. [Online]. Available: http://doi.acm.org/10.1145/3092703.3098222

[38] "Travis ci," https://travis-ci.org/, 2017.

[39] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 426–437.

[40] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, 2015, pp. 805–816.

[41] M. Vasic, Z. Parvez, A. Milicevic, and M. Gligoric, "File-level vs. module-level regression test selection for .net," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 848–853. [Online]. Available: http://doi.acm.org/10.1145/3106237.3117763

[42] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139