Soundy Automated Parallelization of Test Execution

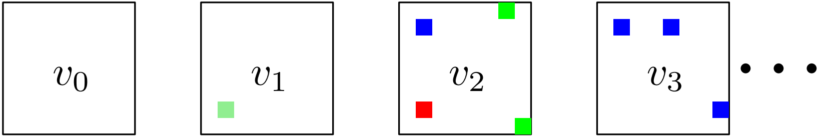
Shouvick Mondal, Denini Silva, Marcelo d'Amorim

IIT Madras (India), UFPE (Brazil), UFPE (Brazil)

ICSME 2021 (Virtual Event)

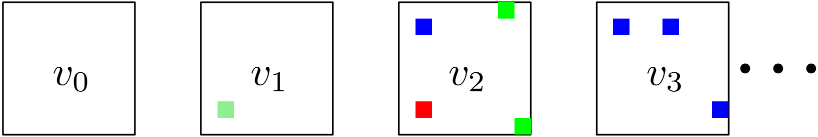
September 27 – October 1

Context: software evolution and regression testing



Regression testing: testing software changes for regression bugs.

Context: software evolution and regression testing



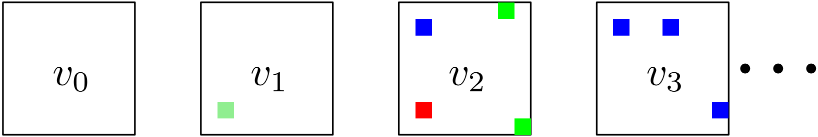
Regression testing: testing software changes for regression bugs.

Existing solutions

Regression Test Selection (RTS)[1]

[1] **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

Context: software evolution and regression testing



Regression testing: testing software changes for regression bugs.

Existing solutions

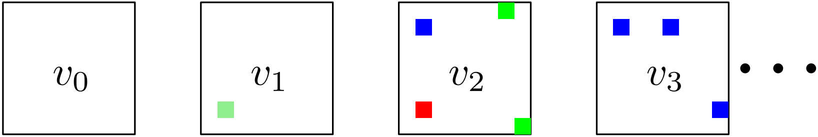
Regression Test Selection (RTS)[1]

Regression Test Prioritization (RTP)[2]

[1] **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

[2] **Source**: S. Elbaum et al., *Test Case Prioritization: A Family of Empirical Studies*, TSE 2002.

Context: software evolution and regression testing



Regression testing: testing software changes for regression bugs.

Existing solutions

Regression Test Selection (RTS)[1]

Regression Test Prioritization (RTP)[2]

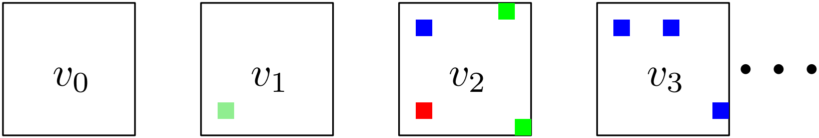
Test Suite Reduction (TSR)[3]

[1] **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

[2] **Source**: S. Elbaum et al., *Test Case Prioritization: A Family of Empirical Studies*, TSE 2002.

[3] **Source**: G. Rothermel et al., *Empirical Studies of Test-Suite Reduction.*, STVR 2002.

Context: software evolution and regression testing



Regression testing: testing software changes for regression bugs.

Existing solutions

Regression Test Selection (RTS)[1]

Regression Test Prioritization (RTP)[2]

Test Suite Reduction (TSR)[3]

Test execution parallelization (*is less explored*...).[4]

[1] **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

[2] **Source**: S. Elbaum et al., *Test Case Prioritization: A Family of Empirical Studies*, TSE 2002.

[3] **Source**: G. Rothermel et al., *Empirical Studies of Test-Suite Reduction.*, STVR 2002.

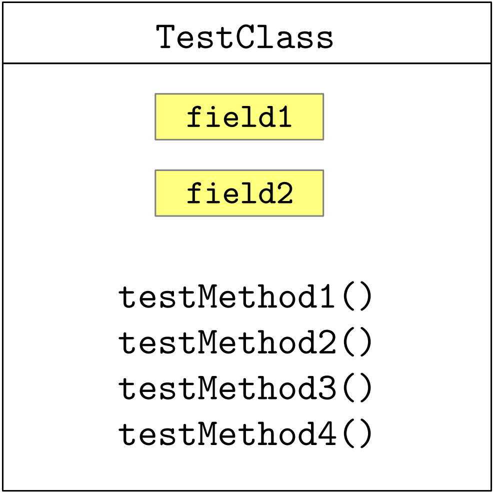
[4] **Source**: J. Candido et al., *Test suite parallelization in open-source projects: A study on its usage and impact.*, ASE 2017.

Two issues in test parallelization

Test dependencies and **data races** give rise to **test flakiness**.

Two issues in test parallelization

Test dependencies and **data races** give rise to **test flakiness**.



TestClass

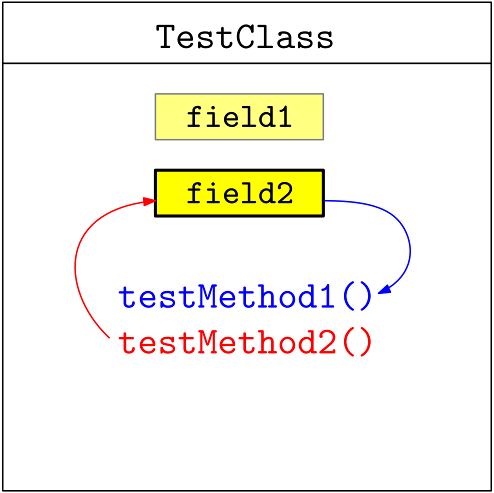
field1

field2

testMethod1()
testMethod2()
testMethod3()
testMethod4()

Two issues in test parallelization

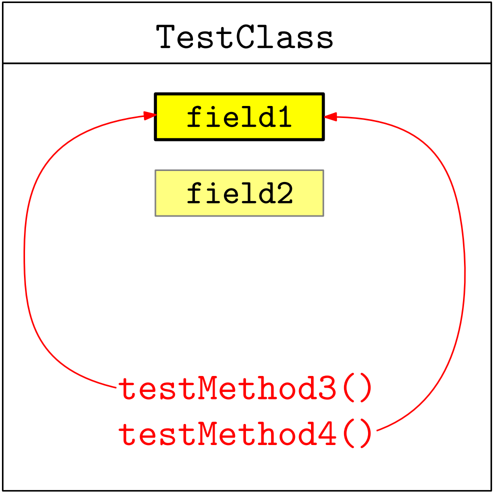
Test dependencies and **data races** give rise to **test flakiness**.



Dependencies: $\{t_1 \to t_2\}$

Two issues in test parallelization

Test dependencies and ***data races*** give rise to **test flakiness**.

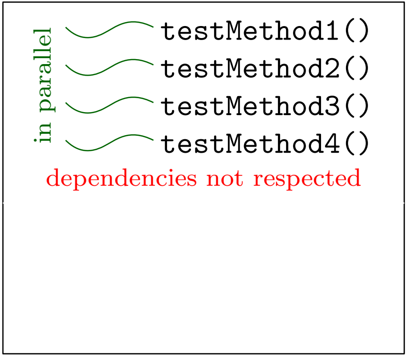


Dependencies: $\{t_4 \rightarrow t_3\}$

Two issues in test parallelization

Test dependencies and **data races** give rise to **test flakiness**.

$$\{t_1 \rightarrow t_2,\ t_4 \rightarrow t_3\}$$

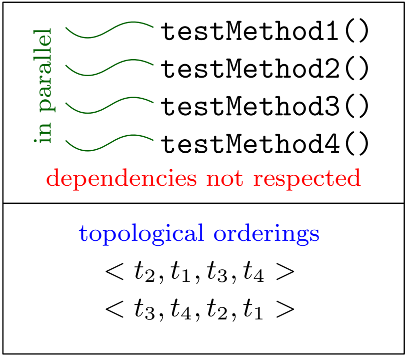


Run 1:	P	P	P	P
Run 2:	P	F	P	P
Run 3:	F	F	P	P
Run 4:	P	P	F	P
Run 5:	F	P	P	F

Two issues in test parallelization

Test dependencies and **data races** give rise to **test flakiness**.

$\{t_1 \rightarrow t_2,\ t_4 \rightarrow t_3\}$



in parallel

testMethod1()

testMethod2()

testMethod3()

testMethod4()

dependencies not respected

topological orderings

$< t_2, t_1, t_3, t_4 >$

$< t_3, t_4, t_2, t_1 >$

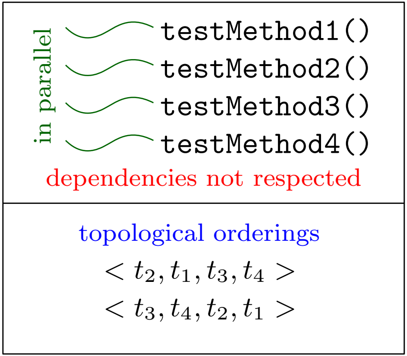
Run 1:	P	P	P	P
Run 2:	P	F	P	P
Run 3:	F	F	P	P
Run 4:	P	P	F	P
Run 5:	F	P	P	F

A topological sort would reveal a **safe** execution sequence.

Two issues in test parallelization

Test dependencies and **data races** give rise to **test flakiness**.

$\{t_1 \rightarrow t_2,\ t_4 \rightarrow t_3\}$



in parallel

`testMethod1()`

`testMethod2()`

`testMethod3()`

`testMethod4()`

dependencies not respected

topological orderings

$< t_2, t_1, t_3, t_4 >$

$< t_3, t_4, t_2, t_1 >$

Run 1:	P	P	P	P
Run 2:	P	F	P	P
Run 3:	F	F	P	P
Run 4:	P	P	F	P
Run 5:	F	P	P	F

A topological sort would reveal a **safe** execution sequence.
But **prerequisite** is *test dependency detection*!

Performance of a test dependency detector

State-of-the-art tool: PRADET (ICST 2018)

PRADET

Step 1 (costs *x*): Sequential execution

Performance of a test dependency detector

State-of-the-art tool: PRADET (ICST 2018)

PRADET

Step 1 (costs x): Sequential execution
Step 2 (costs y): Dynamic data-flow analysis

Performance of a test dependency detector

State-of-the-art tool: PRADET (ICST 2018)

PRADET

- **Step 1** (costs *x*): Sequential execution
- **Step 2** (costs *y*): Dynamic data-flow analysis
- **Step 3** (costs *z*): Dependency refinement

Performance of a test dependency detector

State-of-the-art tool: PRADET (ICST 2018)

PRADET

Step 1 (costs x): Sequential execution
Step 2 (costs y): Dynamic data-flow analysis
Step 3 (costs z): Dependency refinement

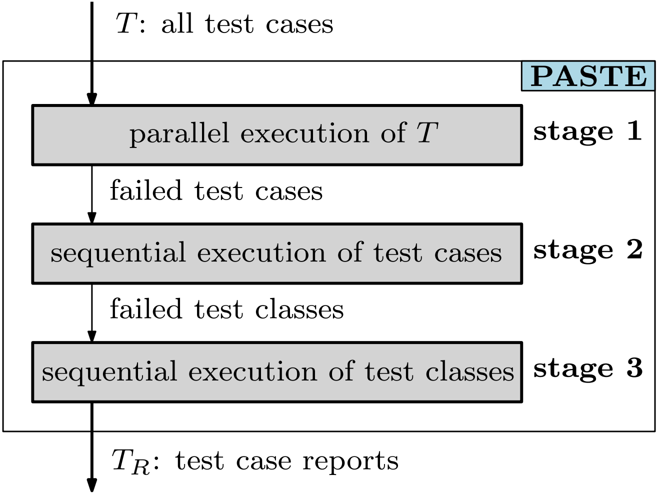
The overhead of PRADET was **substantially higher than sequential execution itself** ($y + z > x$).
NOT practical *to use PRADET* to aid test parallelization!

Our approach: **PASTE**

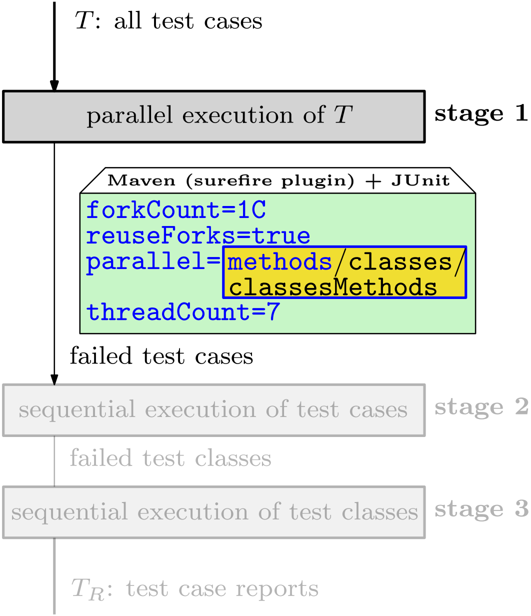
PArallel-**S**equential **T**est **E**xecution

Our approach: **PASTE**

PArallel-**S**equential **T**est **E**xecution



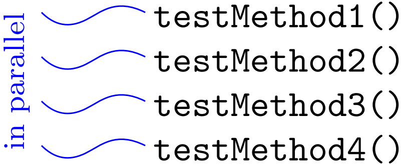
Stage 1: parallel execution



T: all test cases
parallel execution of T
stage 1
Maven (surefire plugin) + JUnit
failed test cases
sequential execution of test cases
stage 2
failed test classes
sequential execution of test classes
stage 3
T_R: test case reports

```
forkCount=1C
reuseForks=true
parallel=methods/classes/
         classesMethods
threadCount=7
```

Stage 1: parallel execution



in parallel

testMethod1()

testMethod2()

testMethod3()

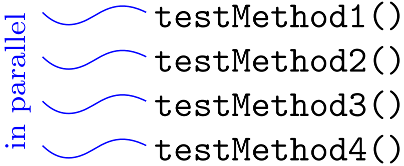
testMethod4()

dependencies not respected

$$\{t_1 \rightarrow t_2,\ t_4 \rightarrow t_3\}$$

Execute the four test methods **in parallel**.

Stage 1: parallel execution



in parallel

`testMethod1()`

`testMethod2()`

`testMethod3()`

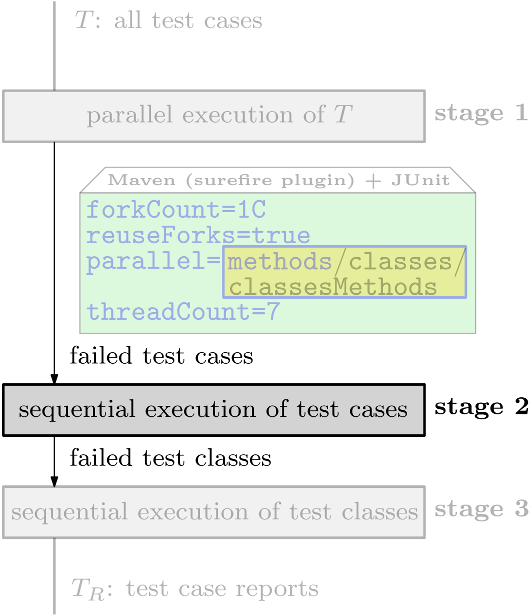
`testMethod4()`

dependencies not respected

$\{t_1 \rightarrow t_2,\ t_4 \rightarrow t_3\}$

Execute the four test methods **in parallel**.
Some test cases may fail!

Stage 2: sequential re-execution of failed test cases



T: all test cases

parallel execution of T

stage 1

Maven (surefire plugin) + JUnit

```
forkCount=1C
reuseForks=true
parallel=methods/classes/
         classesMethods
threadCount=7
```

failed test cases

sequential execution of test cases

stage 2

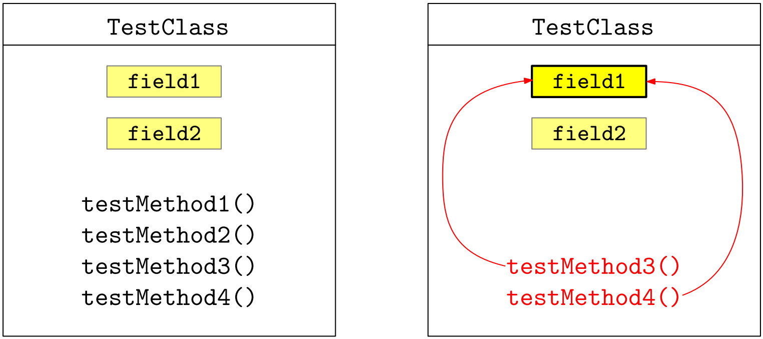
failed test classes

sequential execution of test classes

stage 3

T_R: test case reports

Stage 2: sequential re-execution of failed test cases



TestClass

field1

field2

testMethod1()
testMethod2()
testMethod3()
testMethod4()

TestClass

field1

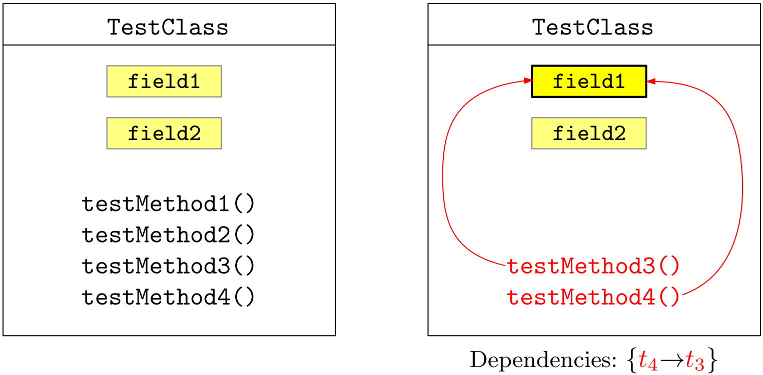
field2

testMethod3()
testMethod4()

Dependencies: $\{t_4 \rightarrow t_3\}$

Handle flakiness through **sequential** **re-execution** of **test cases** (*to circumvent **data races***).

Stage 2: sequential re-execution of failed test cases

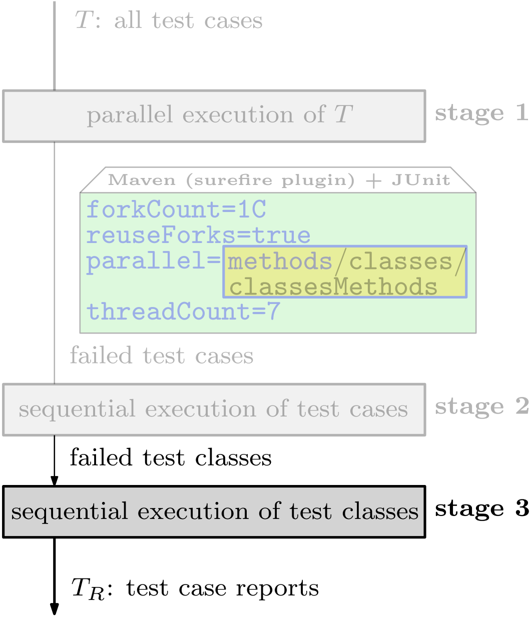


Dependencies: $\{t_4 \rightarrow t_3\}$

Handle flakiness through **sequential** **re-execution** of **test cases** (*to circumvent* ***data races***).

Some test cases may fail ***again***!
Track their test class names.

Stage 3: sequential re-execution of failed test classes



T: all test cases

parallel execution of T

stage 1

Maven (surefire plugin) + JUnit

```
forkCount=1C
reuseForks=true
parallel=methods/classes/
         classesMethods
threadCount=7
```

failed test cases

sequential execution of test cases

stage 2

failed test classes

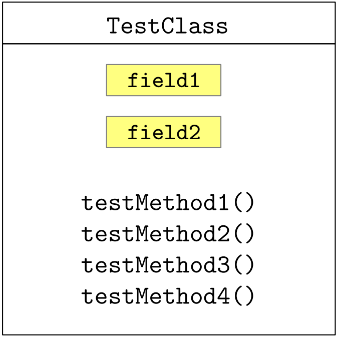
sequential execution of test classes

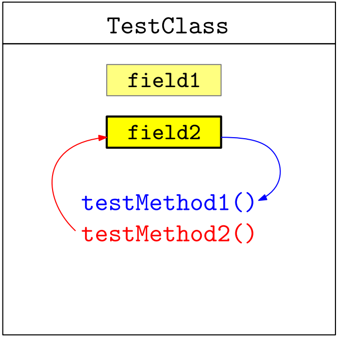
stage 3

T_R: test case reports

Stage 3: sequential re-execution of failed test classes

Handle flakiness through **sequential re-execution** of **test classes** (*to circumvent **broken test dependencies***).

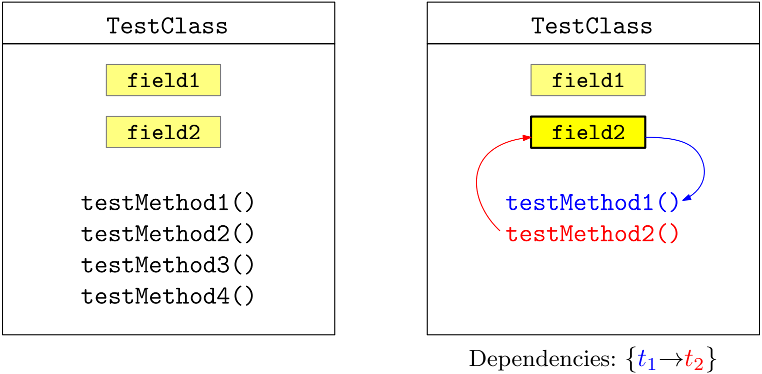




Dependencies: $\{t_1 \rightarrow t_2\}$

Stage 3: sequential re-execution of failed test classes

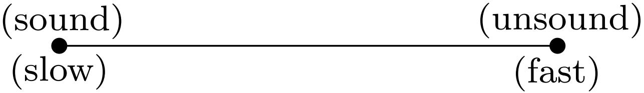
Handle flakiness through **sequential re-execution** of **test classes** (*to circumvent **broken test dependencies***).



PASTE builds on the observation:
broken test dependencies that are manifested in parallel runs ***involve test cases from the same test class***.

The spectrum of soundness in parallelization

Sound: time invariant verdicts agree with sequential execution.



(sound)

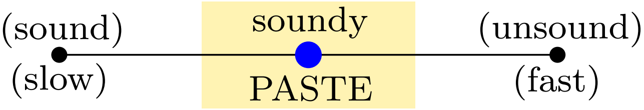
(slow)

(unsound)

(fast)

The spectrum of soundness in parallelization

Sound: time invariant verdicts agree with sequential execution.



PASTE does not provide the soundness guarantee but is reasonable enough to yield end-to-end acceleration!

Experimental Setup

Experimental Setup

Hardware: **8 CPUs** (**4 cores**, with **2 threads per core**).

Experimental Setup

Hardware: **8 CPUs** (**4 cores**, with **2 threads per core**).

Software: GNU **Bash** 5.0.17, and **Maven** 3.6.3.

Experimental Setup

Hardware: **8 CPUs** (**4 cores**, with **2 threads per core**).

Software: GNU **Bash** 5.0.17, and **Maven** 3.6.3.

Subjects: **25 Java projects** that use **Maven** and have at least 200 stars and 300 tests.

Experimental Setup

Hardware: **8 CPUs** (**4 cores**, with **2 threads per core**).

Software: GNU **Bash** 5.0.17, and **Maven** 3.6.3.

Subjects: **25 Java projects** that use **Maven** and have at least 200 stars and 300 tests.

No failures: Reran each test suite 10 times to **identify** and **eliminate tests failing** due to **non-determinism**.

Research Questions

RQ1 (feasibility #1)

Is it feasible to use parallelization options provided by the build system "out of the box" to run test suites?

RQ1 (feasibility #1)

Is it feasible to use parallelization options provided by the build system "out of the box" to run test suites?

In 44% of the projects, no parallel configurations enabled a clean execution. Searching for the **parallel configuration for a clean execution is INFEASIBLE in general**.

RQ2 (feasibility #2)

Is it practical to use a test dependency analyzer to partition test sets as to enable sound parallel execution?

RQ2 (feasibility #2)

Is it practical to use a test dependency analyzer to partition test sets as to enable sound parallel execution?

The runtime overhead of PRADET was substantially higher than that of the sequential execution itself. **NOT PRACTICAL to use PRADET to aid test parallelization**.

RQ3 (effectiveness #1)

How reliable is* PASTE *?

RQ3 (effectiveness #1)

How reliable is **PASTE** ***?***

Effective to circumvent the test flakiness provoked by test parallelization. There were **no cases of provoked failure that "survived" the third stage** of PASTE.

RQ4 (effectiveness #2)

What are the speedups obtained with* PASTE*?

RQ4 (effectiveness #2)

What are the speedups obtained with* PASTE*?

We observed speedups in 52% of the projects. The **configuration `classes`** performed the best: **median 1.59x** (best: 2.28x, average: 1.47x, worst: 0.93x).

Related Work

Most relevant related work

Work–venue	Description	Relation
ElectricTest–FSE 2015	Tracks test dependencies on global resources + distributed parallelization.	Complementary

Most relevant related work

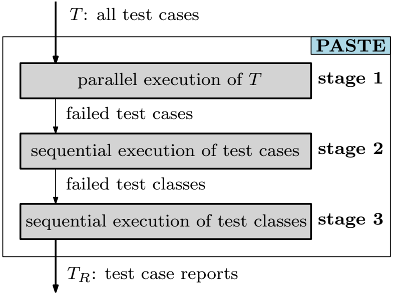
Work–venue	Description	Relation
ElectricTest–FSE 2015	Tracks test dependencies on global resources + distributed parallelization.	Complementary
ParTeCL–ISSTA 2017	Transforms code to GPU kernels for GPU level parallelization.	Domain specific Yet to explore

Most relevant related work

Work–venue	Description	Relation
ElectricTest–FSE 2015	Tracks test dependencies on global resources + distributed parallelization.	Complementary
ParTeCL–ISSTA 2017	Transforms code to GPU kernels for GPU level parallelization.	Domain specific Yet to explore
TEDD–FSE 2019	NLP-based web test dependency detector tracks client-server network operations.	Domain specific Yet to explore

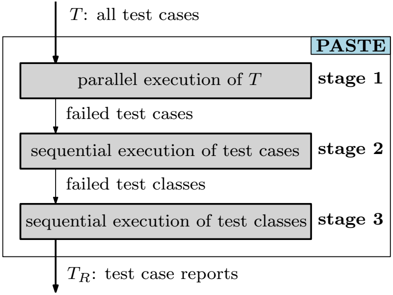
Conclusions

We discussed **PASTE**, a lightweight approach to parallelize execution of test suites through the sequential re-execution of *test cases* (*to avoid data races*) and the sequential re-execution of *test classes* (*to avoid broken test dependencies*).



Conclusions

We discussed **PASTE**, a lightweight approach to parallelize execution of test suites through the sequential re-execution of *test cases* (*to avoid data races*) and the sequential re-execution of *test classes* (*to avoid broken test dependencies*).



T: all test cases

PASTE

parallel execution of T — **stage 1**

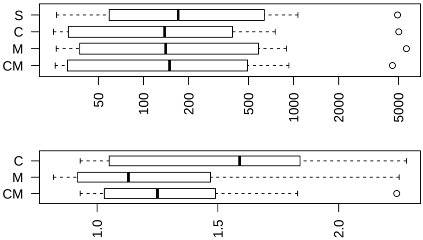
failed test cases

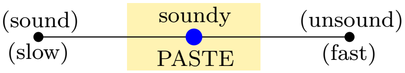
sequential execution of test cases — **stage 2**

failed test classes

sequential execution of test classes — **stage 3**

T_R: test case reports

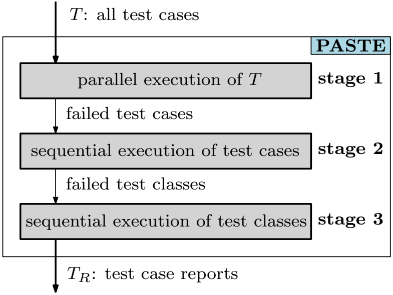


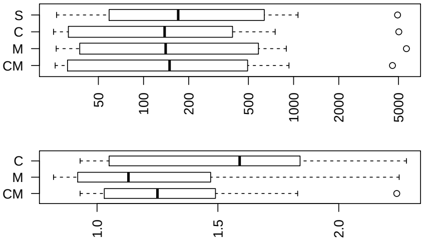


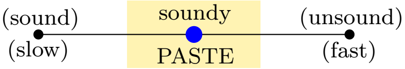
(sound) (slow) — soundy PASTE — (unsound) (fast)

Conclusions

We discussed **PASTE**, a lightweight approach to parallelize execution of test suites through the sequential re-execution of *test cases* (*to avoid data races*) and the sequential re-execution of *test classes* (*to avoid broken test dependencies*).







Thank You

Artifacts: `https://github.com/STAR-RG/paste`