Soundy Automated Parallelization of Test Execution

Shouvick Mondal, Denini Silva, Marcelo d'Amorim

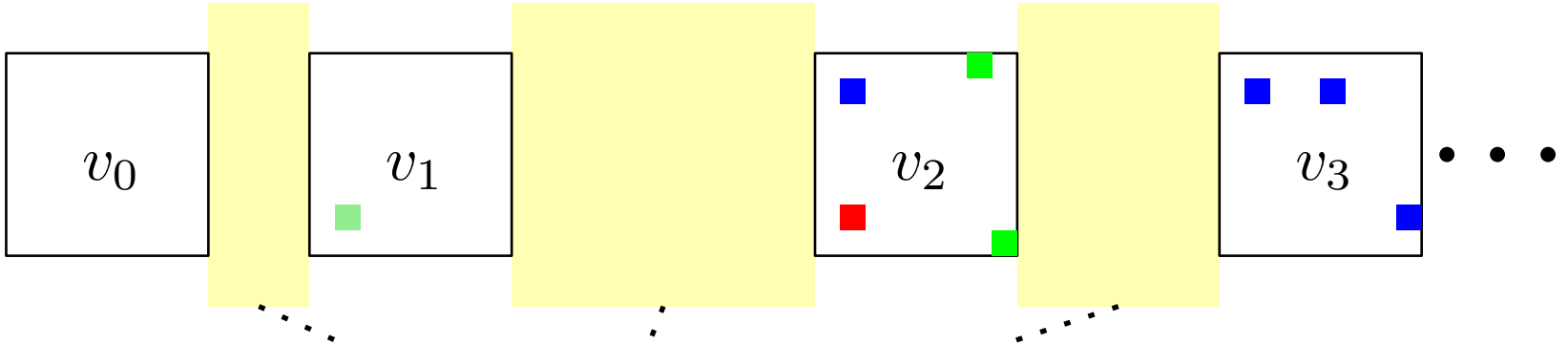
IIT Madras (India), UFPE (Brazil), UFPE (Brazil)



ICSME 2021 (Virtual Event)

September 27 – October 1

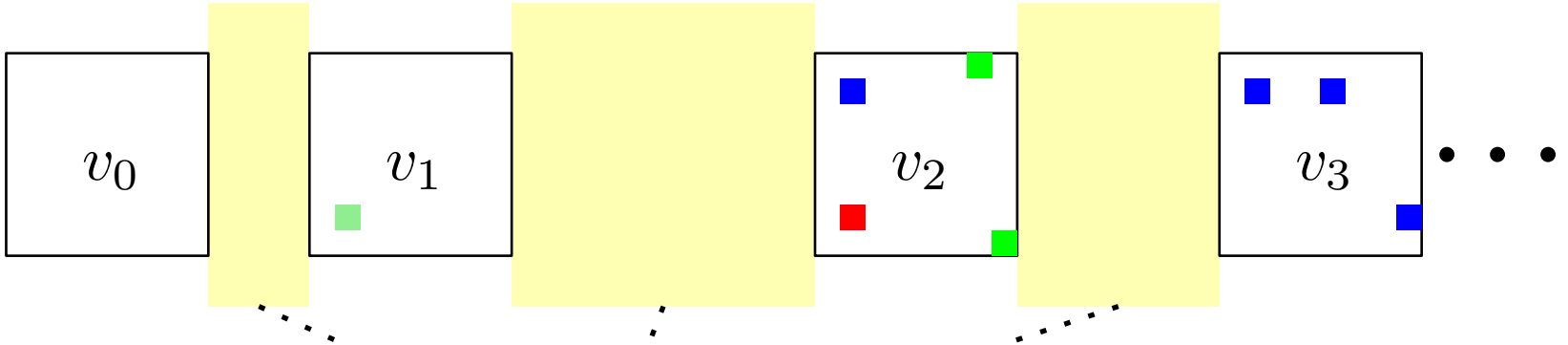
Context: software evolution and regression testing



Time between updates is small + changes need to be tested

Regression testing efficiency is important!

Context: software evolution and regression testing

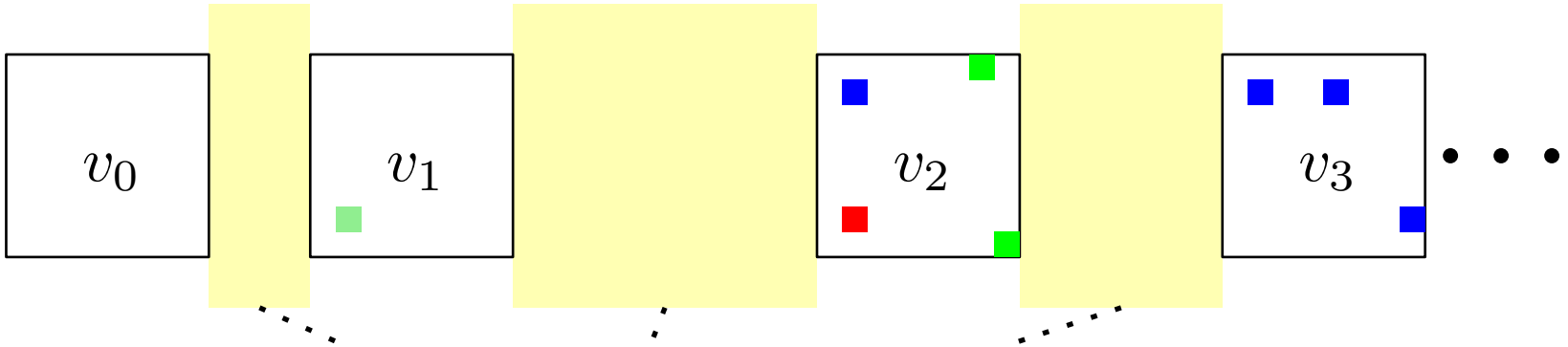


Time between updates is small + changes need to be tested

Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

Context: software evolution and regression testing



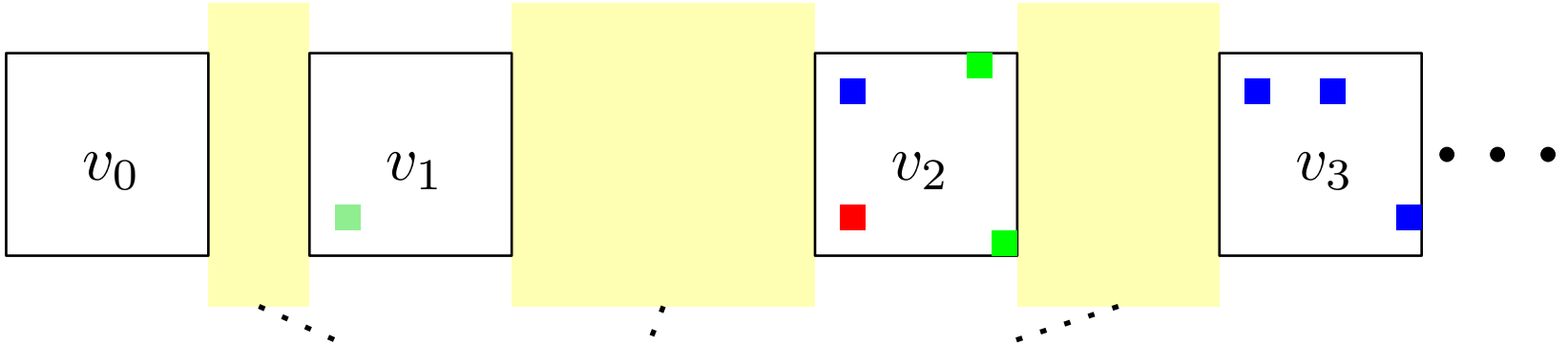
Time between updates is small + changes need to be tested
Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

Industrial example: RetestAll took seven weeks to complete![1]

[1] **Source**: S. Elbaum et al., *Prioritizing Test Cases for Regression Testing*, ISSTA 2000.

Context: software evolution and regression testing



Time between updates is small + changes need to be tested
Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

Industrial example: RetestAll took seven weeks to complete![1]

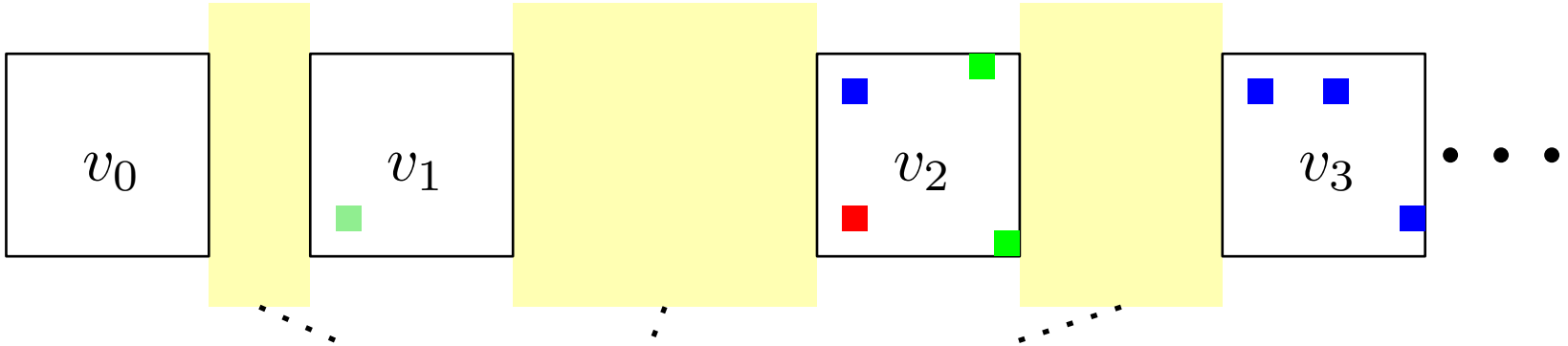
Sophisticated solutions

- Regression test selection (RTS).[2]

[1] **Source**: S. Elbaum et al., *Prioritizing Test Cases for Regression Testing*, ISSTA 2000.

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

Context: software evolution and regression testing



Time between updates is small + changes need to be tested
Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

Industrial example: RetestAll took seven weeks to complete![1]

Sophisticated solutions

- Regression test selection (RTS).[2]
- Regression test prioritization (RTP).[3]

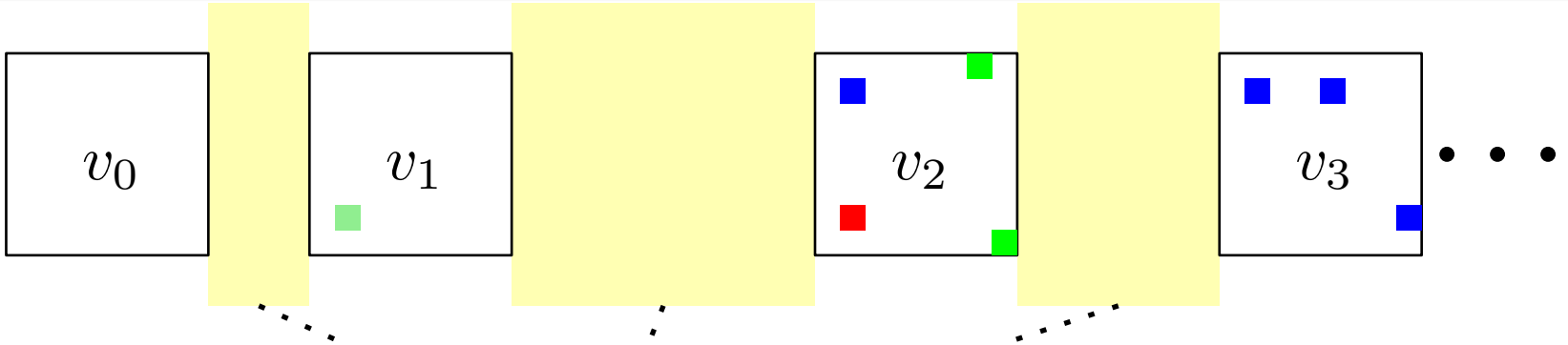
---

1 **Source**: S. Elbaum et al., *Prioritizing Test Cases for Regression Testing*, ISSTA 2000.

2 **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

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

Context: software evolution and regression testing



Time between updates is small + changes need to be tested
Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

Industrial example: RetestAll took seven weeks to complete![1]

Sophisticated solutions

- Regression test selection (RTS).[2]
- Regression test prioritization (RTP).[3]
- Test-suite reduction (TSR).[4]

---

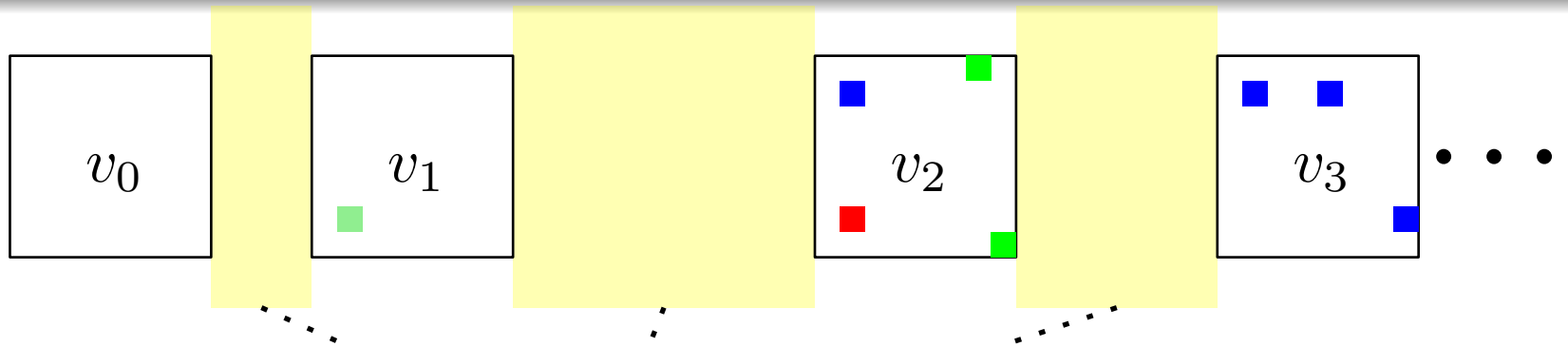
1 **Source**: S. Elbaum et al., *Prioritizing Test Cases for Regression Testing*, ISSTA 2000.

2 **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.

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

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

Context: software evolution and regression testing



Time between updates is small + changes need to be tested
Regression testing efficiency is important!

RetestAll: execute all test-cases sequentially? (*expensive*!)

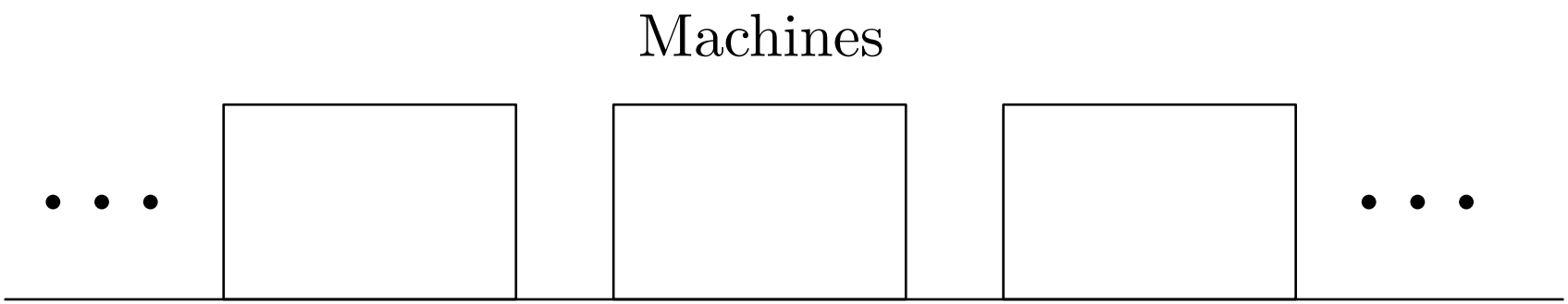
Industrial example: RetestAll took seven weeks to complete![1]

Sophisticated solutions

- Regression test selection (RTS).[2]
- Regression test prioritization (RTP).[3]
- Test-suite reduction (TSR).[4]
- ***Test-execution parallelization*** (*is less explored*...).[5]

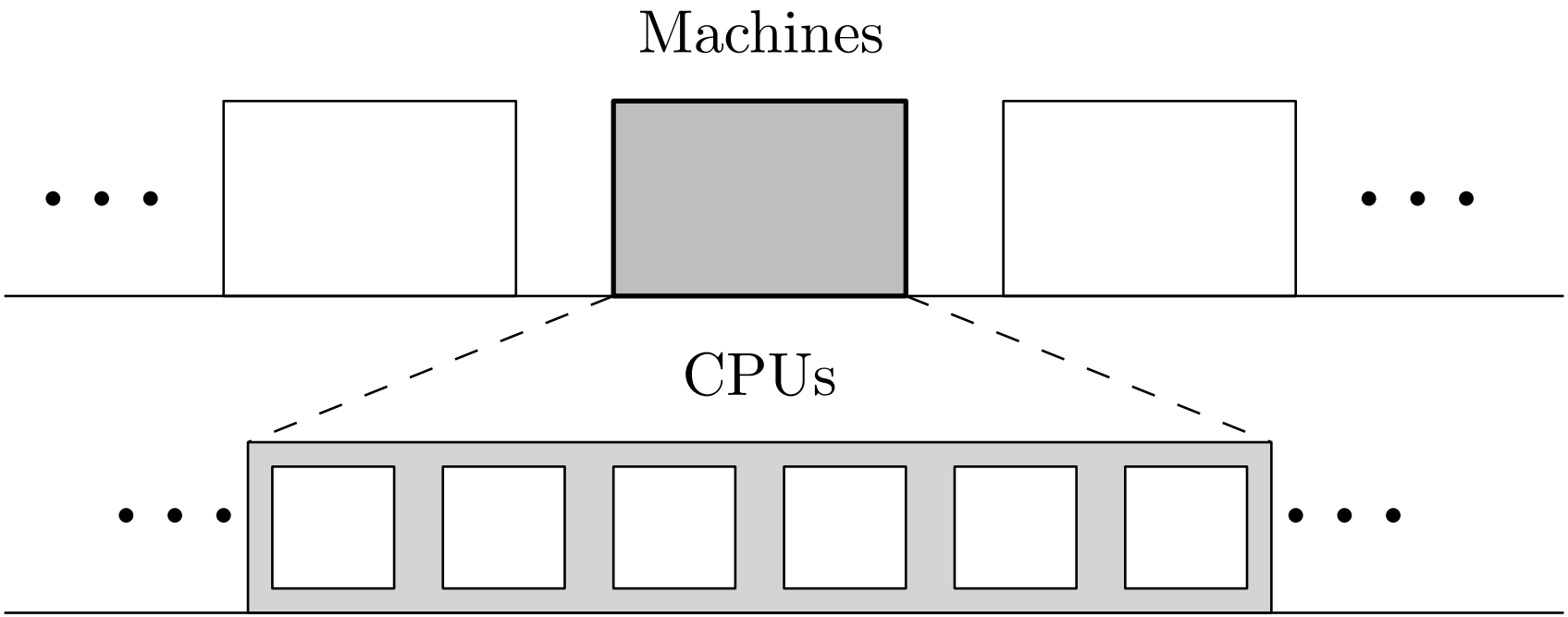
1 **Source**: S. Elbaum et al., *Prioritizing Test Cases for Regression Testing*, ISSTA 2000.
2 **Source**: M. Gligoric et al., *Ekstazi: Lightweight Test Selection*, ICSE 2015.
3 **Source**: S. Elbaum et al., *Test Case Prioritization: A Family of Empirical Studies*, TSE 2002.
4 **Source**: G. Rothermel et al., *Empirical Studies of Test-Suite Reduction.*, STVR 2002.
5 **Source**: J. Candido et al., *Test suite parallelization in open-source projects: A study on its usage and impact.*, ASE 2017.

Test parallelization levels



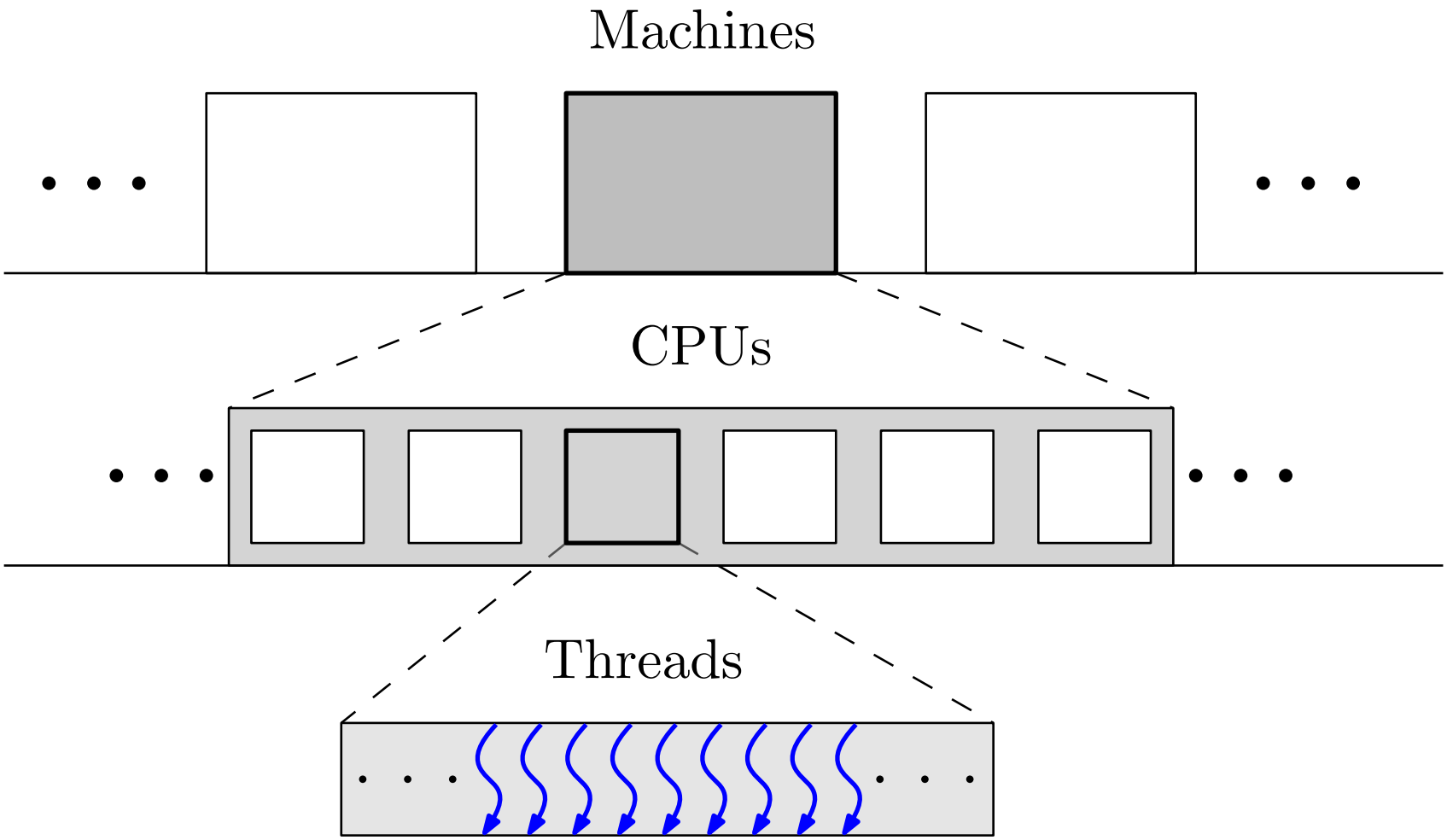
Source: J. Candido et al., *Test suite parallelization in open-source projects: A study on its usage and impact*, ASE 2017.

Test parallelization levels



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

Test parallelization levels

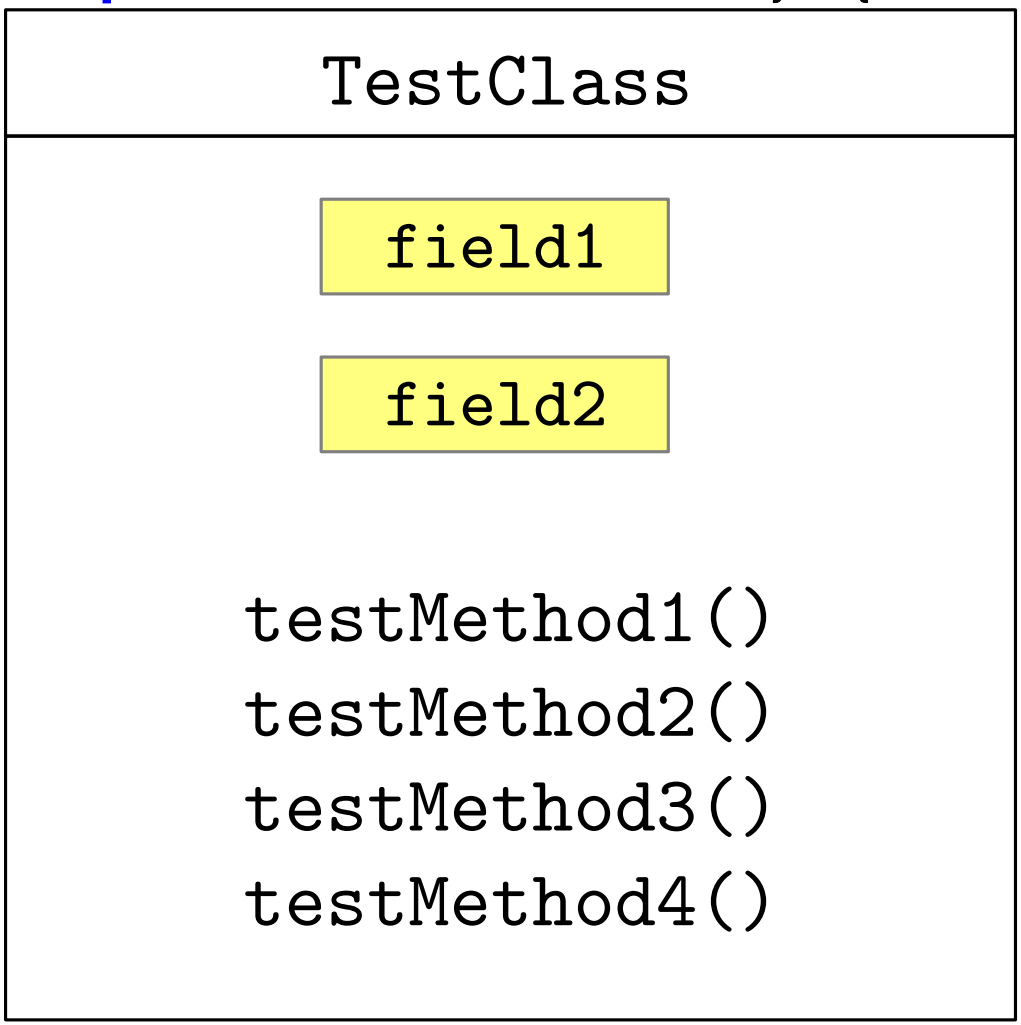


In this work, we focus on *CPU* and *thread* level parallelism.

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

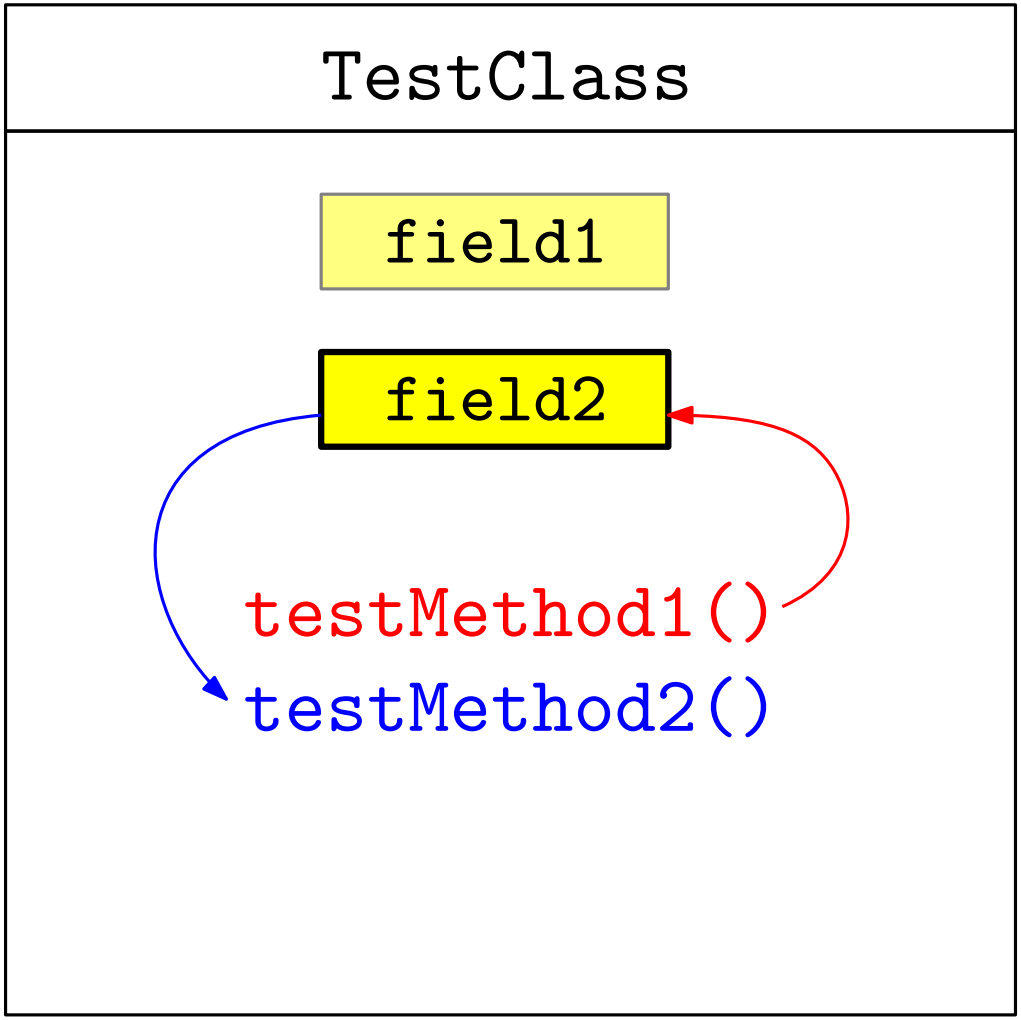
Test flakiness: a major issue due to parallelization

{**Test dependencies**, **Data-races**}→{**flakiness**}



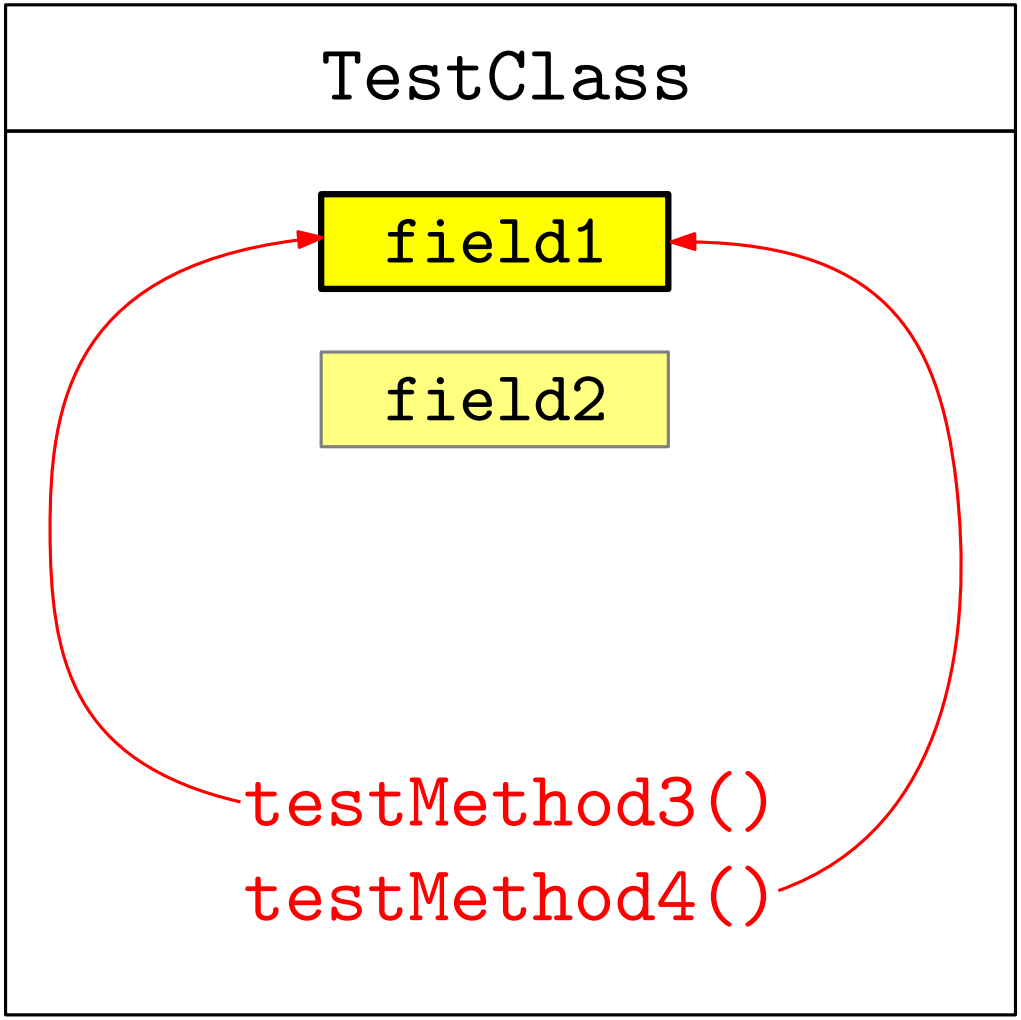
Test flakiness: a major issue due to parallelization

{**Test dependencies**, **Data-races**}→{**flakiness**}



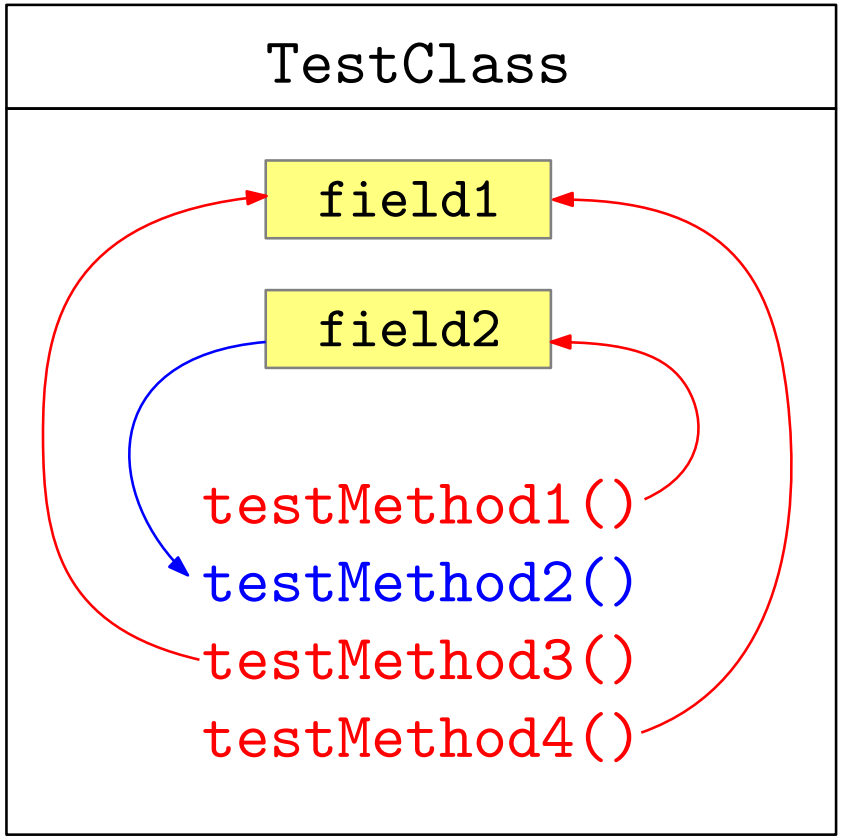
Test flakiness: a major issue due to parallelization

{**Test dependencies**, **Data-races**}→{**flakiness**}



Test flakiness: a major issue due to parallelization

{**Test dependencies**, **Data-races**}→{**flakiness**}



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

Test dependency detection: *can it aid test parallelization*?

Test dependency detection can be costly!

State-of-the-art dependency detector: PRADET (ICST 2018).

PRADET

Step 1 (costs x): Sequential execution to record original test ordering.

Test dependency detection can be costly!

State-of-the-art dependency detector: PRADET (ICST 2018).

PRADET

Step 1 (costs x): Sequential execution to record original test ordering.

Step 2 (costs y): Dynamic data-flow analysis collect data dependencies.

Test dependency detection can be costly!

State-of-the-art dependency detector: PRADET (ICST 2018).

PRADET

Step 1 (costs x): Sequential execution to record original test ordering.

Step 2 (costs y): Dynamic data-flow analysis collect data dependencies.

Step 3 (costs z): Iterative refinement to keep problematic dependencies.

Test dependency detection can be costly!

State-of-the-art dependency detector: PRADET (ICST 2018).

PRADET

Step 1 (costs x): Sequential execution to record original test ordering.

Step 2 (costs y): Dynamic data-flow analysis collect data dependencies.

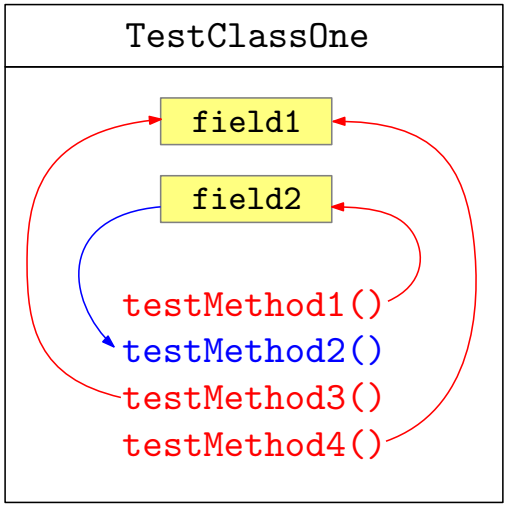
Step 3 (costs z): Iterative refinement to keep problematic dependencies.

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

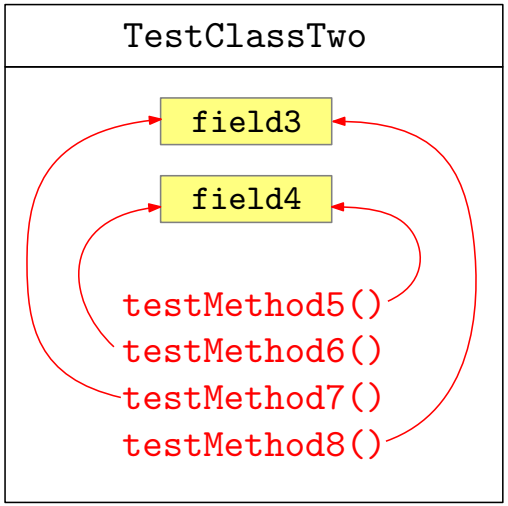
Our approach: **PASTE**
PArallel-**S**equential **T**est **E**xecution

Our approach: **PASTE**

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



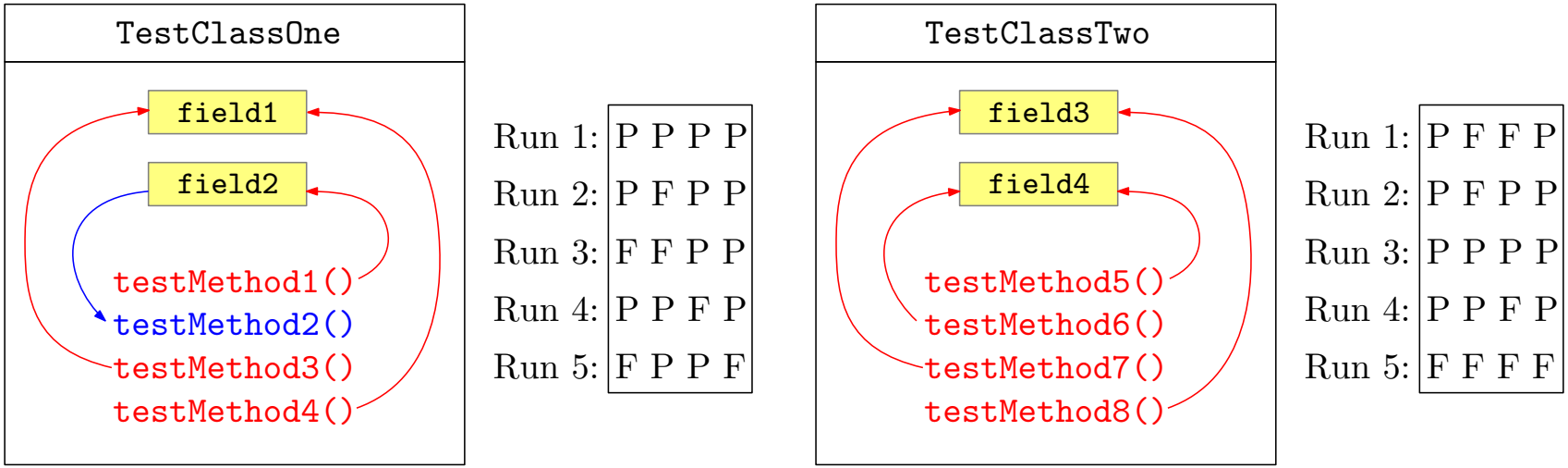
Run 1:	P	P	P	P
Run 2:	P	F	P	P
Run 3:	F	F	P	P
Run 4:	P	P	F	P
Run 5:	F	P	P	F



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

Our approach: **PASTE**

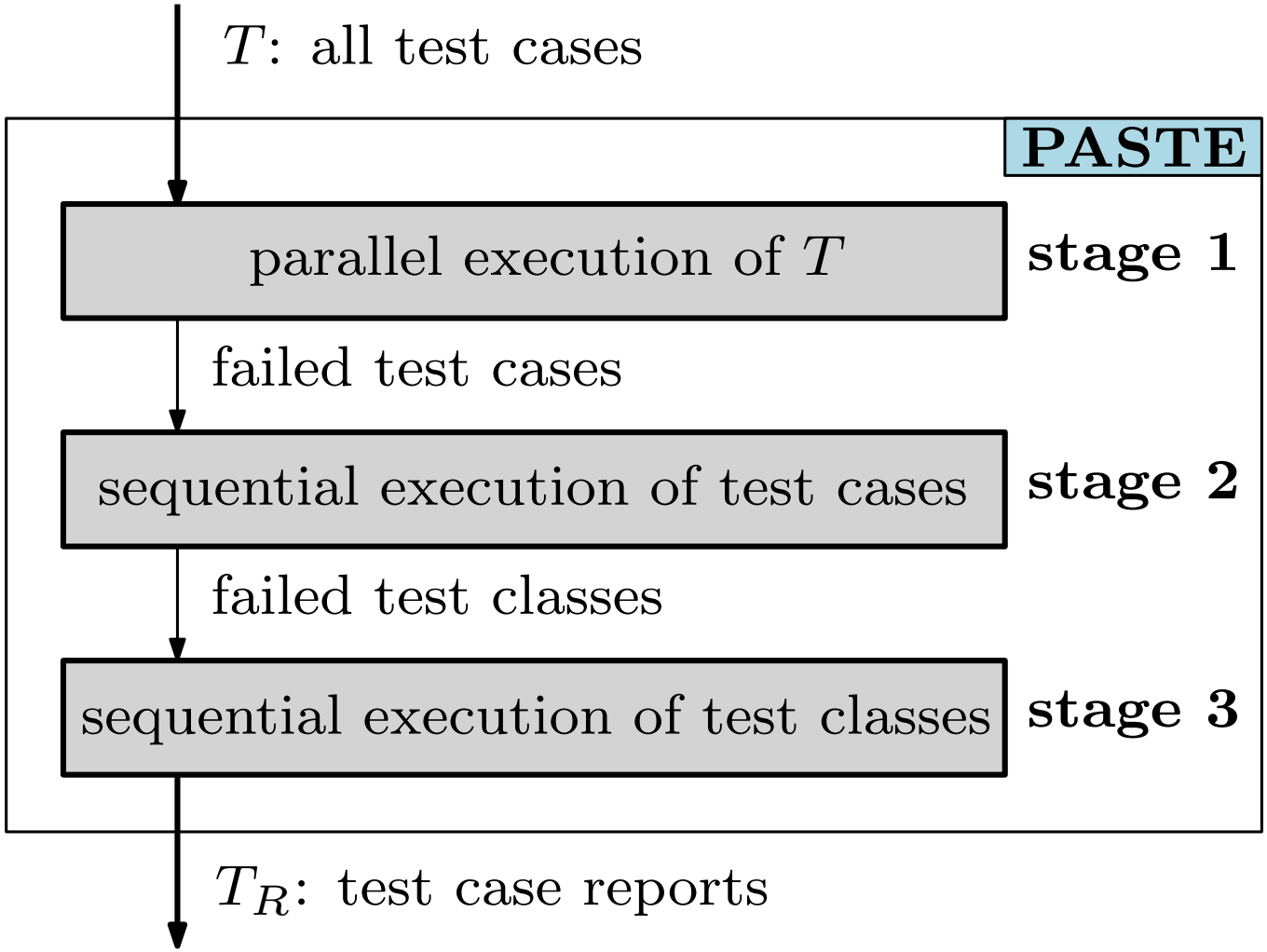
PASTE builds on the observation:
broken test dependencies that are manifested in parallel runs ***involve test cases from the same test class***.



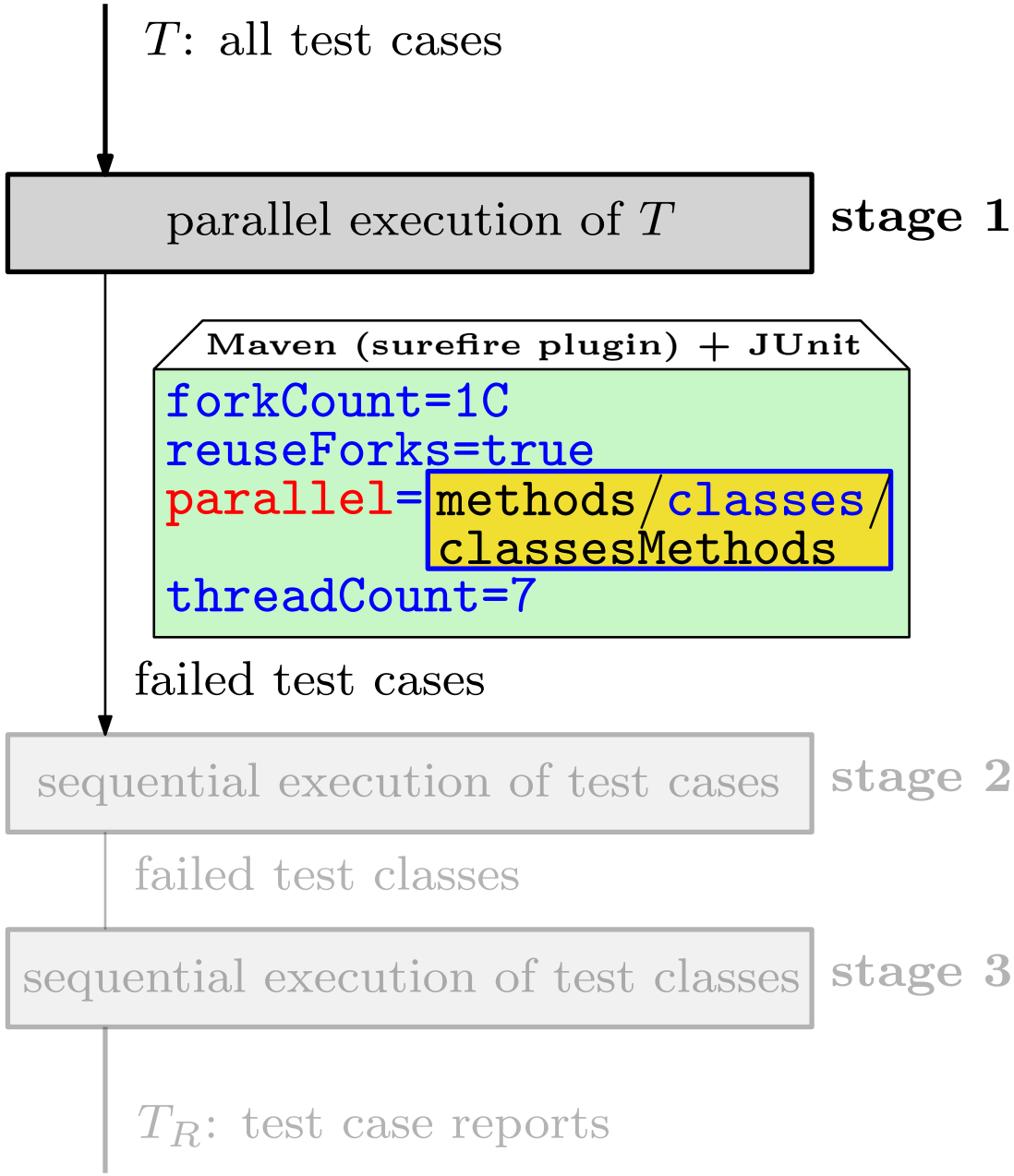
Handle flakiness in test suites through:

- **sequential re-execution** of **test cases**
  (*to avoid* ***data races***).
- **sequential re-execution** of **test classes**
  (*to avoid* ***broken test dependencies***).

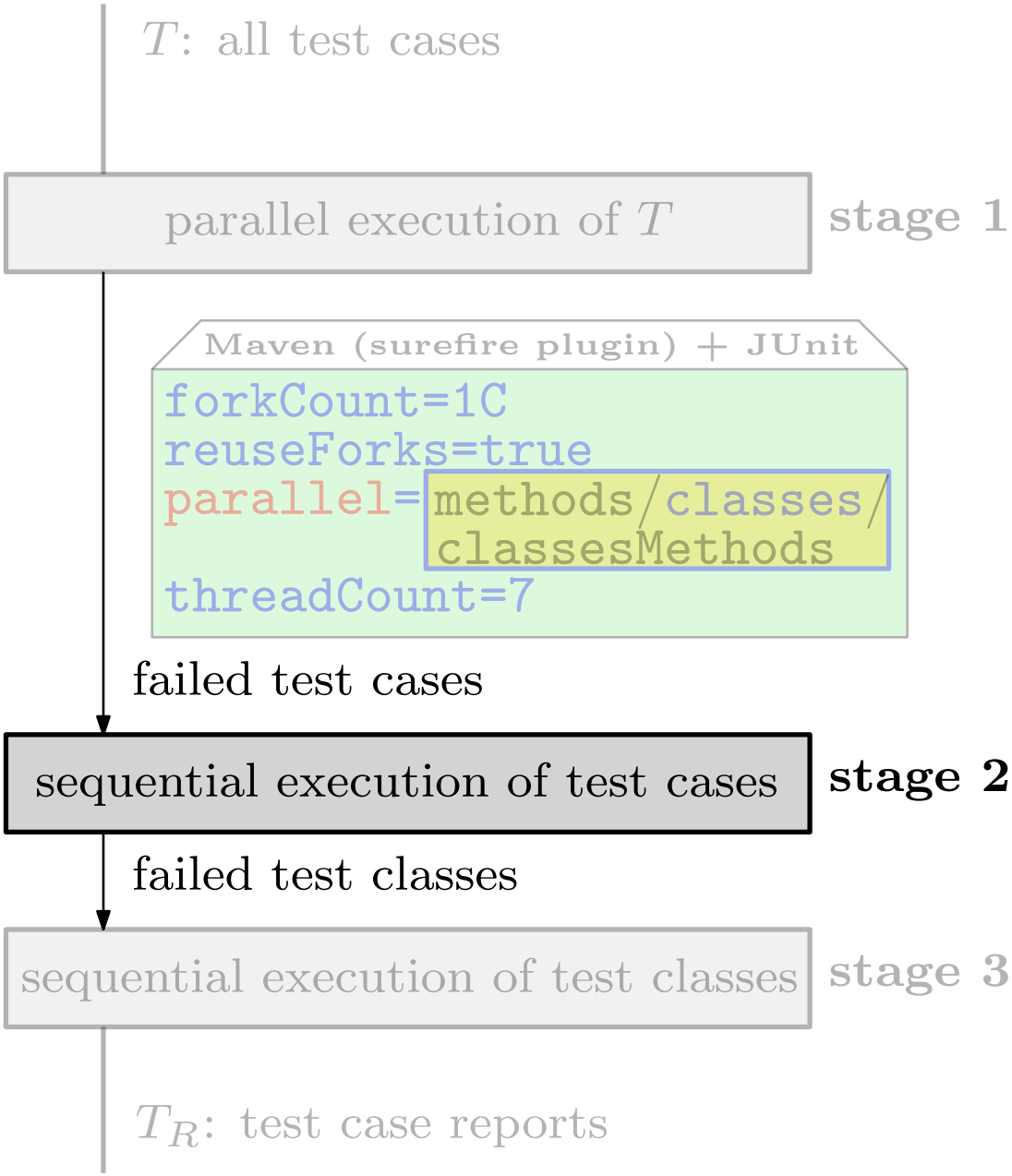
PASTE: the three-staged pipeline of test execution



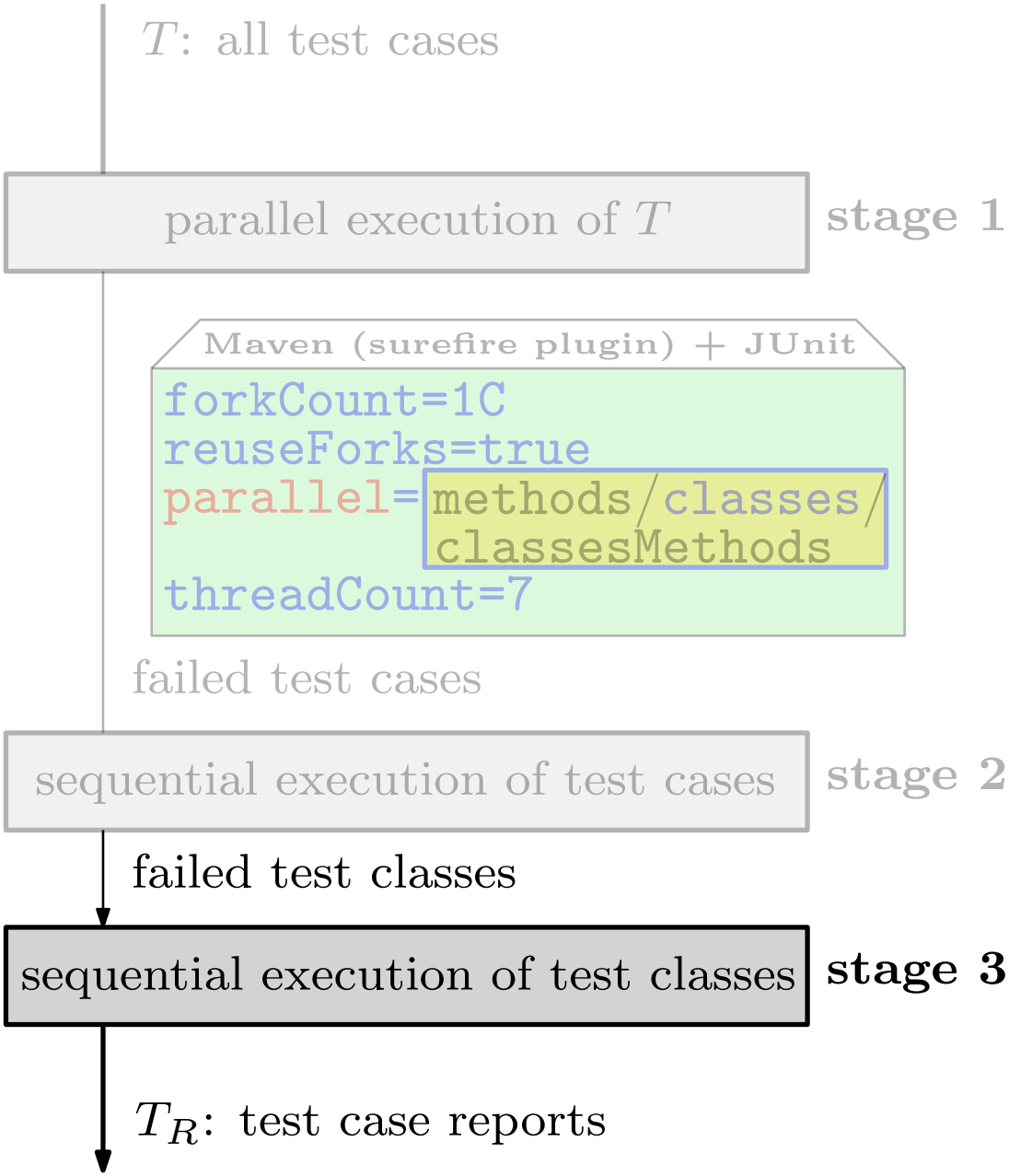
Stage 1: parallel execution



Stage 2: sequential re-execution of failed test cases



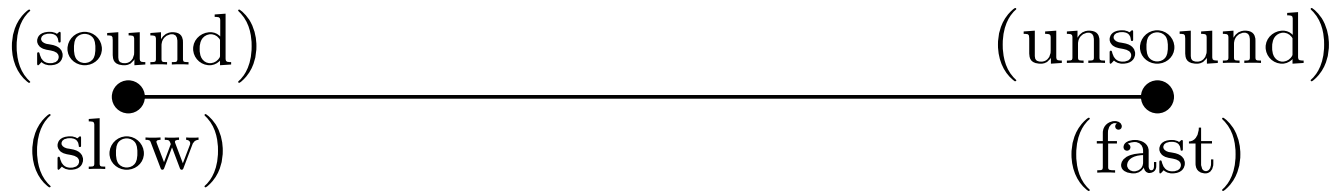
Stage 3: sequential re-execution of failed test classes



The spectrum of soundness in parallelization

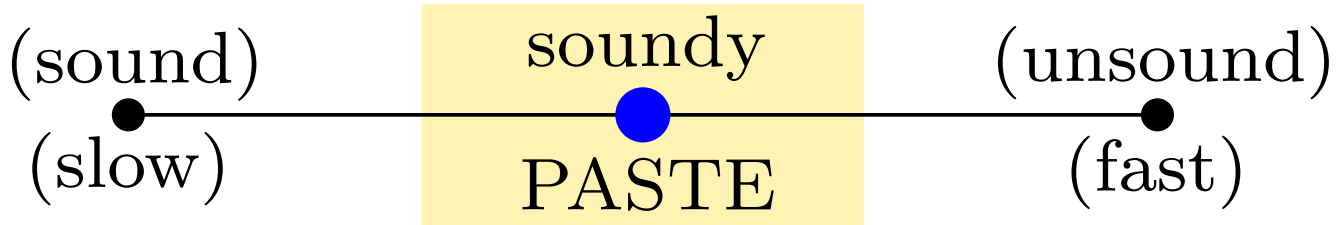
The spectrum of soundness in parallelization

Sound: time invariant verdicts agree with sequential execution.



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**.

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**.
- **Parallel configuration**: parameters in **Maven** and **JUnit**.

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)

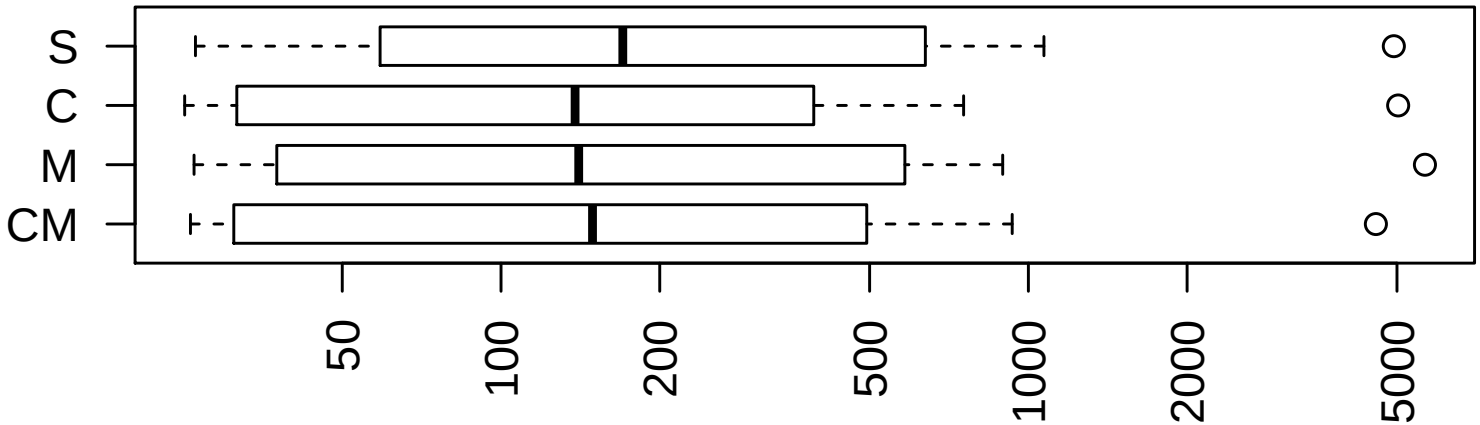
What are the speedups obtained with* PASTE*?

RQ4 (effectiveness #2)

What are the speedups obtained with* PASTE*?

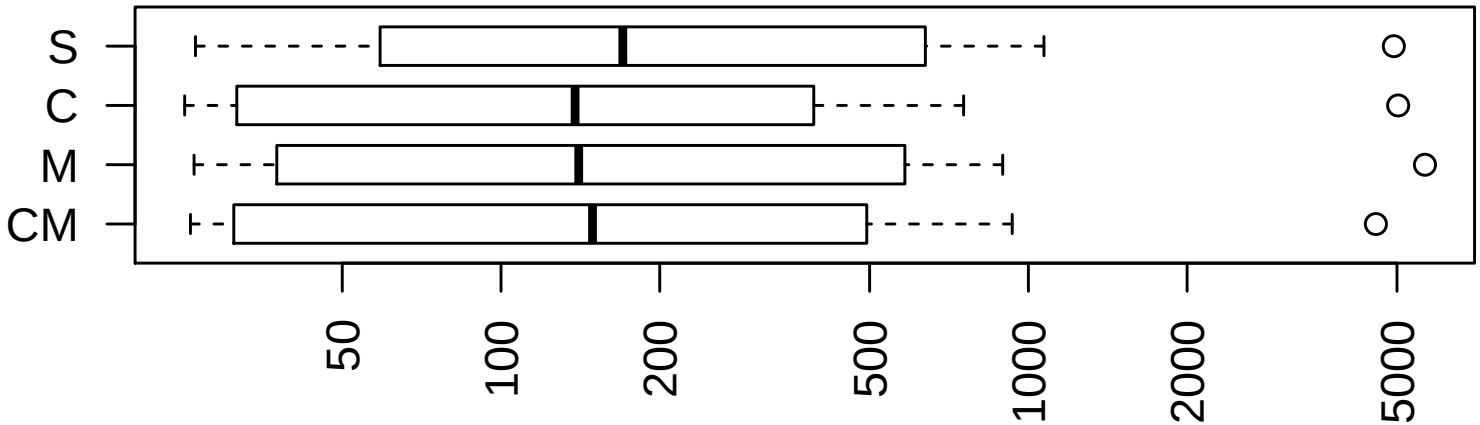
We observed speedups in 52% of the projects. The **configuration `classes`** performed the best: **median 1.59x** (best: 2.28x, average: 1.47x, worst: 0.93x).

RQ4 (effectiveness #2)

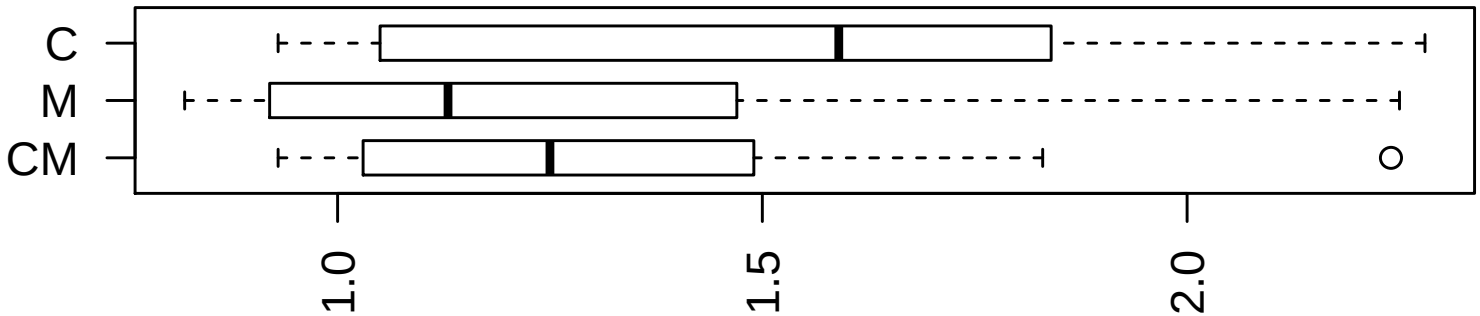


Distribution of **PASTE running times (seconds)** for Sequential (S) and each configuration (Classes (C), Methods (M), ClassesMethods (CM)).

RQ4 (effectiveness #2)



Distribution of **PASTE running times (seconds)** for Sequential (S) and each configuration (Classes (C), Methods (M), ClassesMethods (CM)).



Distribution of **speedups** for each configuration (Classes (C), Methods (M), ClassesMethods (CM)).

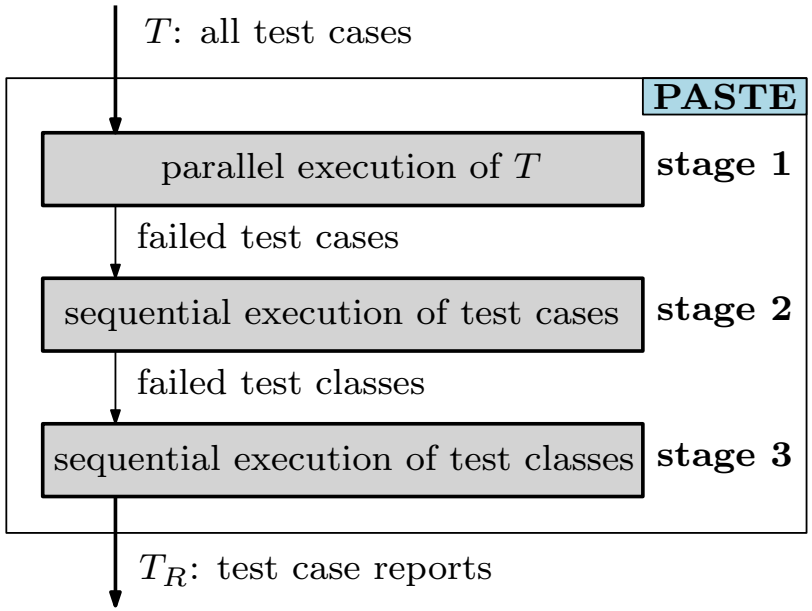
Related Work

Some recent approaches towards test parallelization

Work–venue	Languages used	Speedup	Machine type
ElectricTest–FSE 2015	Java	Avg. 16.00×	Amazon EC2
ParTeCL (GPU)–ISSTA 2017	OpenCL and C (subset)	Avg. 16.00×	GPU
Candido et al.–ASE 2017 (multi-core)	Java	Avg. 3.53× Avg. 4.20×	8 cores @ 3.60 GHz 80 cores @ 2.20 GHz
Mahtab–JSS 2019 (multi-core)	C++, LLVM, and C	Avg. 5.17× G.M. 4.72×	40 cores @ 2.40 GHz 40 cores @ 2.40 GHz

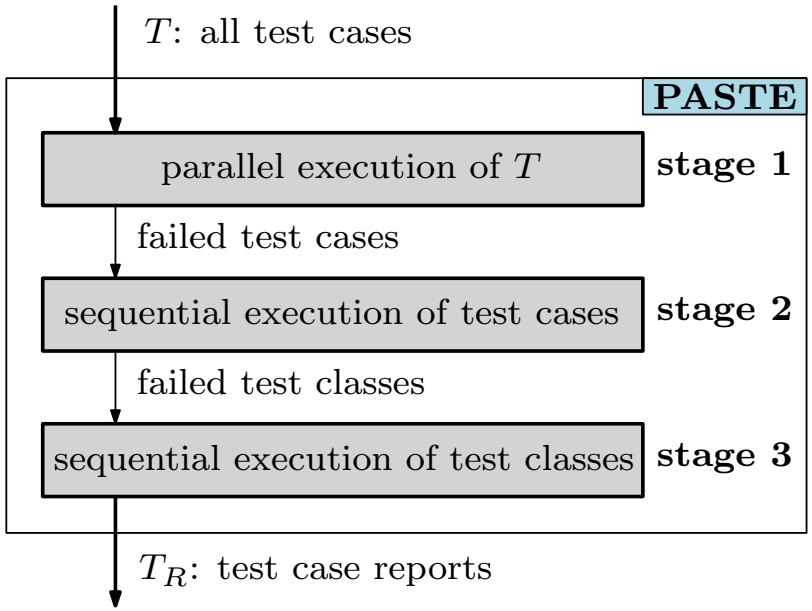
Conclusions

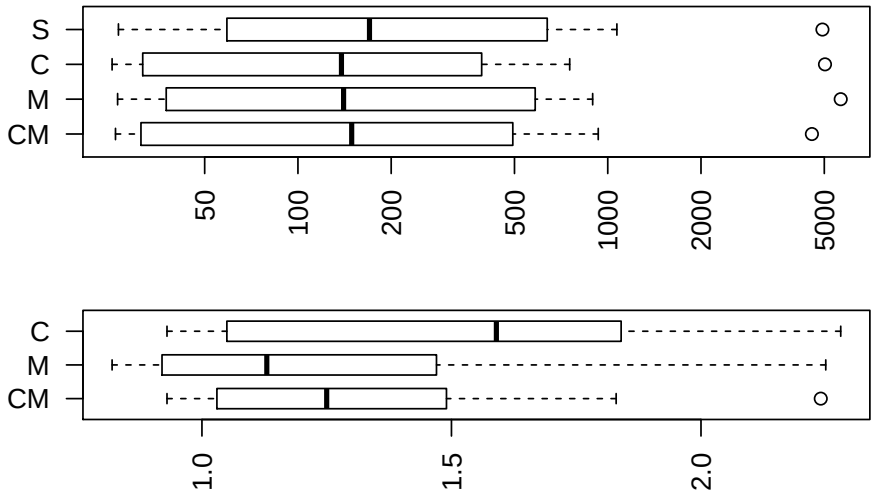
We discussed **PASTE**, a lightweight approach to parallelize execution of test suites through the sequential re-execution of *test cases* (*to avoid data races*) and the sequential re-execution of *test classes* (*to avoid broken test dependencies*).

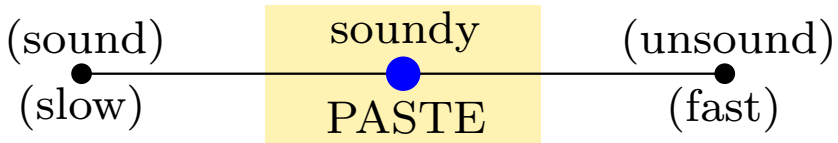


Conclusions

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*).

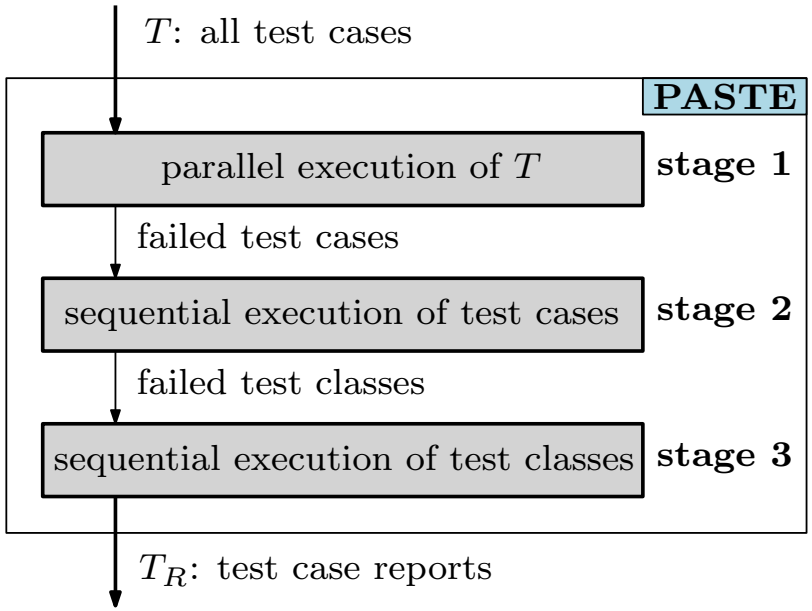


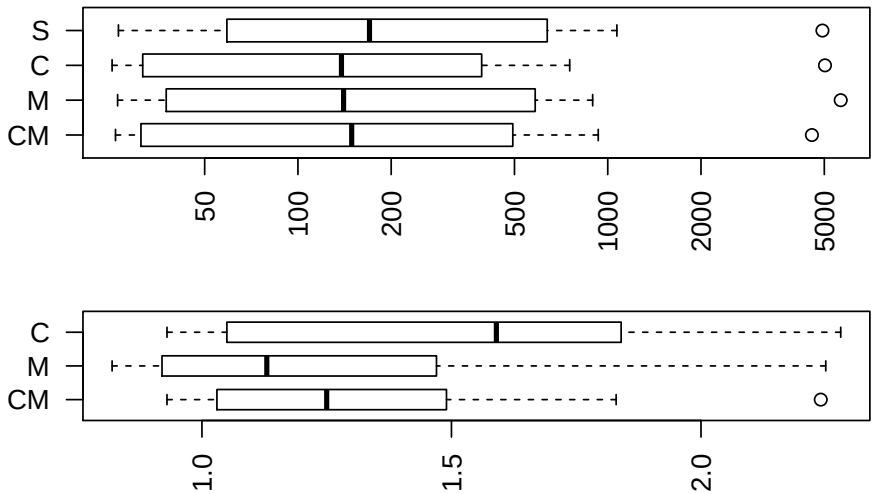


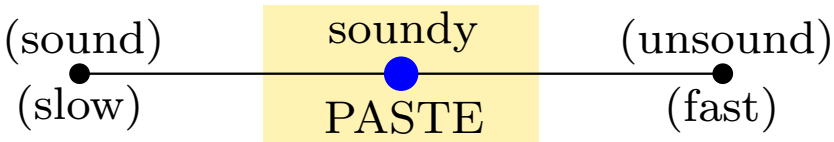


Conclusions

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