

# A Case for Rapid Testing

Rainer Poisel, Stefan Riegler

# About the Speakers

- Rainer Poisel
  - Software Development and Architecture
  - Continuous Integration & Delivery
- Stefan Riegler
  - IIoT, Embedded Security
  - Forensics & Reverse Engineering

# Overview

- Introduction
- State of the Art
- Our Approach
- Results
- Outlook

# Introduction

- Increase Performance of Longer Running Tests
  - Integration and System Tests
- Advantages
  - Faster Feedback
  - Boosting Developer Morale
  - Better CI/CD Usage
  - Increase Coverage

# State of the Art

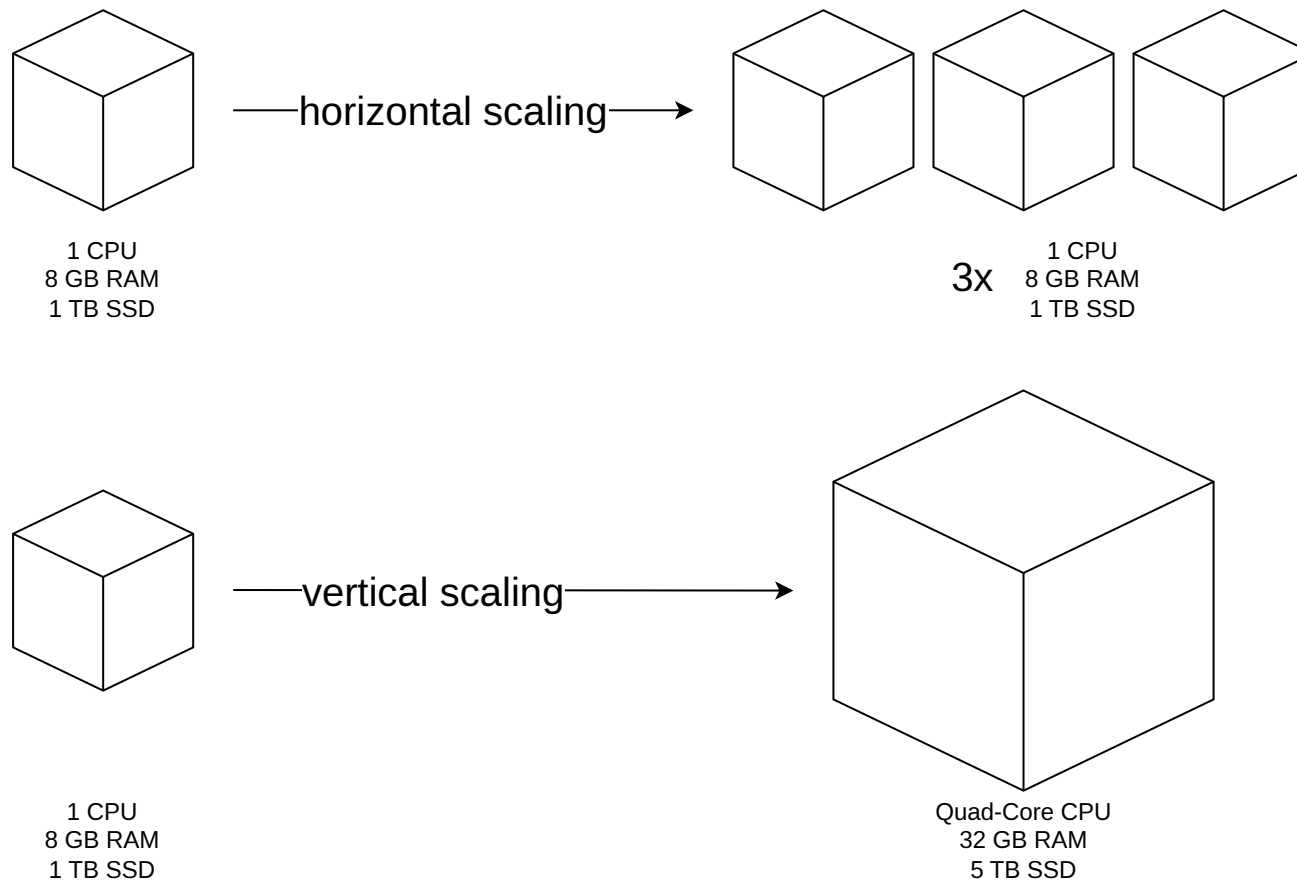
- Test Case Selection and Prioritization
- Improving the Execution Performance
- Parallel Execution of Test Cases

# Selection and Prioritization

- Only execute Subset of Test Cases
  - THEO: Cost-Based Test Selection Strategy
  - Machine-Learning Based Approaches
- How to do that:
  - Consider Test Execution History

# Improving Performance (1)

## Vertical Scaling vs. Horizontal Scaling



# Improving Performance (2)

## Task Parallelism / Task-Parallel Programming

- Work Pool Model
  - Tests dispatched to worker nodes
  - Majority of Worker Nodes Might Remain Idle
- Fork-Join Model
  - Tests explicitly assigned to workers
  - Meticulous Balancing of Test Loads

# Parallelism in Test Frameworks

- JUnit 5 Test Framework:
  - Includes Parallel Execution
  - Orderer Interfaces
- pytest:
  - pytest-parallel (unmaintained)
  - pytest-xdist

# Parallel Execution Efficiency Ratio

- Quotient of
  - $t_s$  ... Duration of Serial Execution
  - $t_p$  ... Duration of Parallel Execution

$$\textit{Efficiency Ratio} = \frac{t_s}{t_p}$$

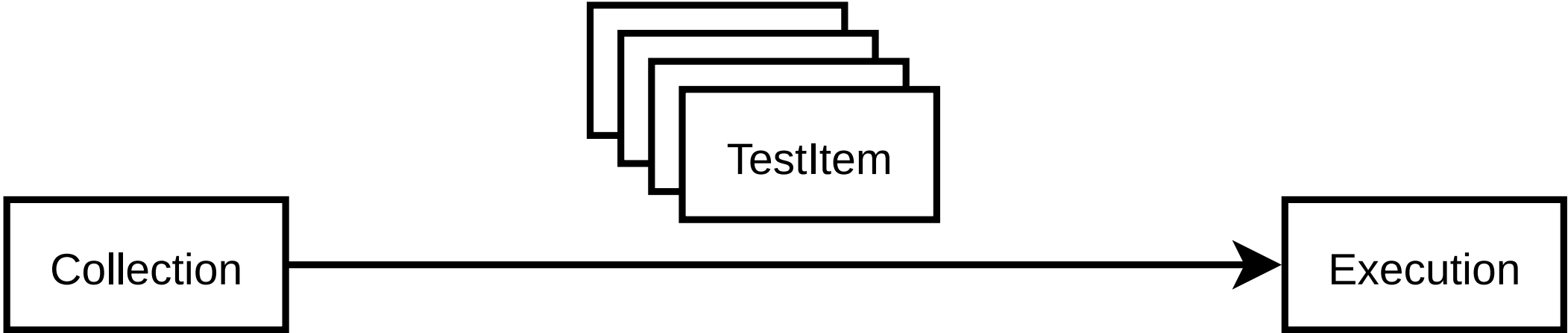
# Improving Test Execution Time

- Approach based on pytest and labgrid frameworks
- Stay compatible with official pytest API
- Utilizing multiple Devices Under Test in Parallel

# Test Suite Overview

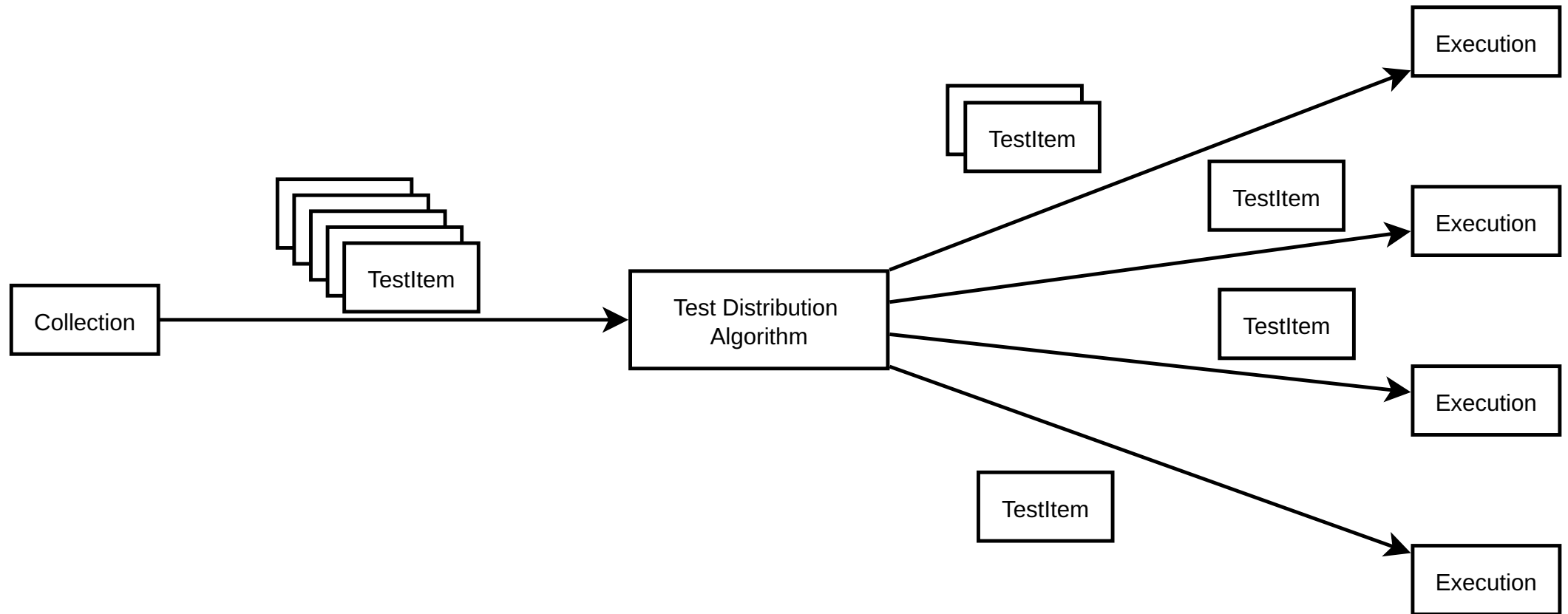
- Test Sets:
  - 1.600 Test Cases
  - 110 Test Cases
- Full Remote Access to Test Infrastructure
- Multiple I/O Subsystems

# Sequential Execution



pytest Test Execution Phases

# Distributing to Multiple Workers



pytest Phases of Parallel Execution

# Our Approach

- Based on the Fork-Join Model
- Simplified One-Dimensional Cutting Stock Solver

# Achieving Even Distribution

- Simplified One-Dimensional Cutting Stock Solver
- Parameters
  - Number of Workers
  - Estimated Test Execution Duration
  - Split Threshold

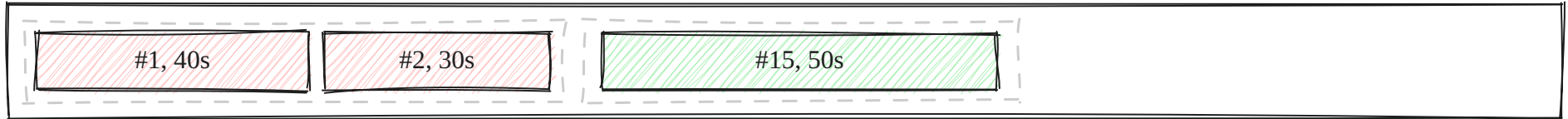
# Assigning Weights to Tests

- Using Python Decorations
- Weights are Specified in Seconds
- Solver supports weighted/unweighted Tests

```
1 @pytest.mark.est_exec_duration(17)
2 def test_configure_parameters(...):
3     ...
```

# 1D Cutting Stock Solver (1)

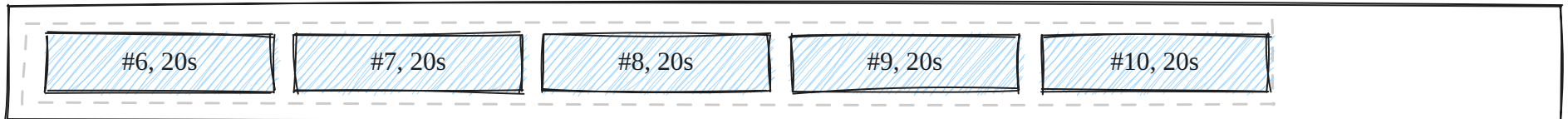
Worker 1



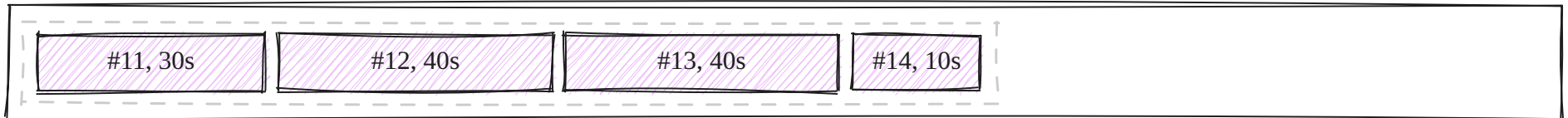
Worker 2



Worker 3

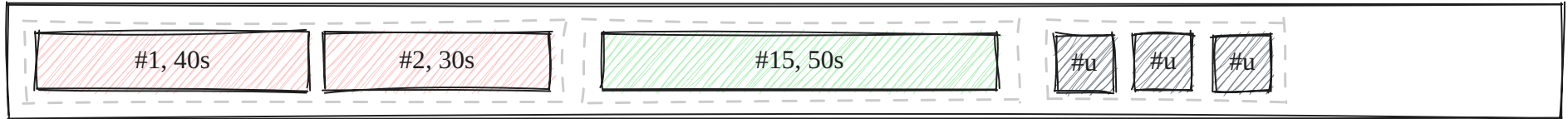


Worker 4



# 1D Cutting Stock Solver (2)

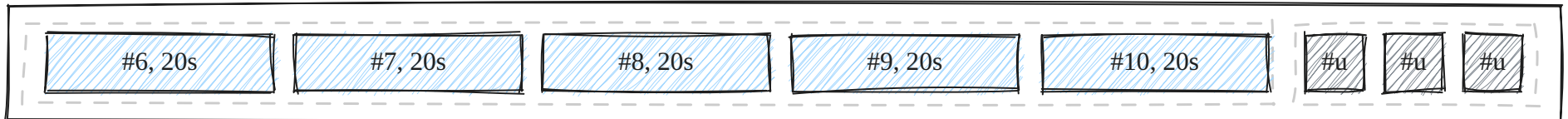
Worker 1



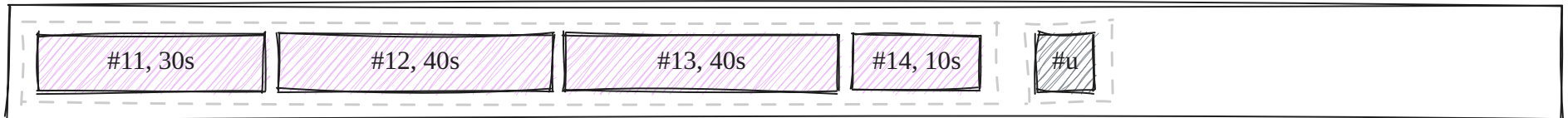
Worker 2



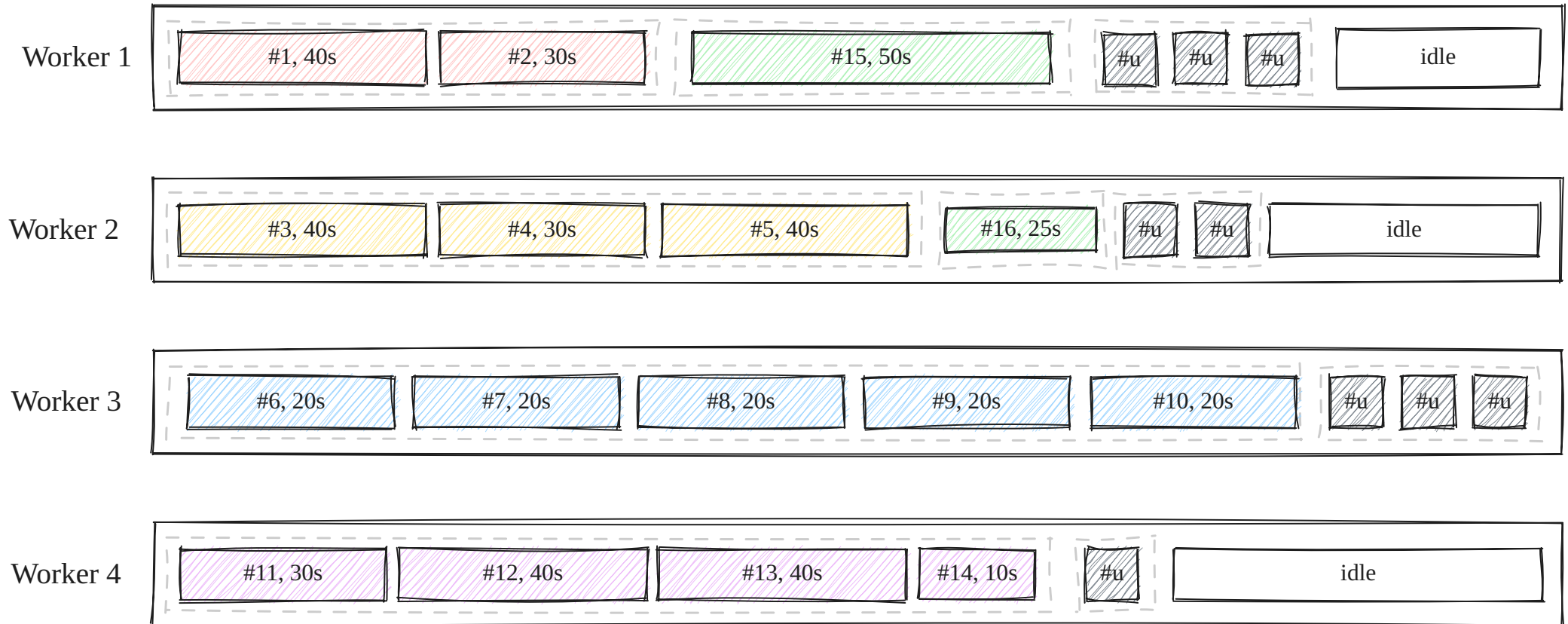
Worker 3



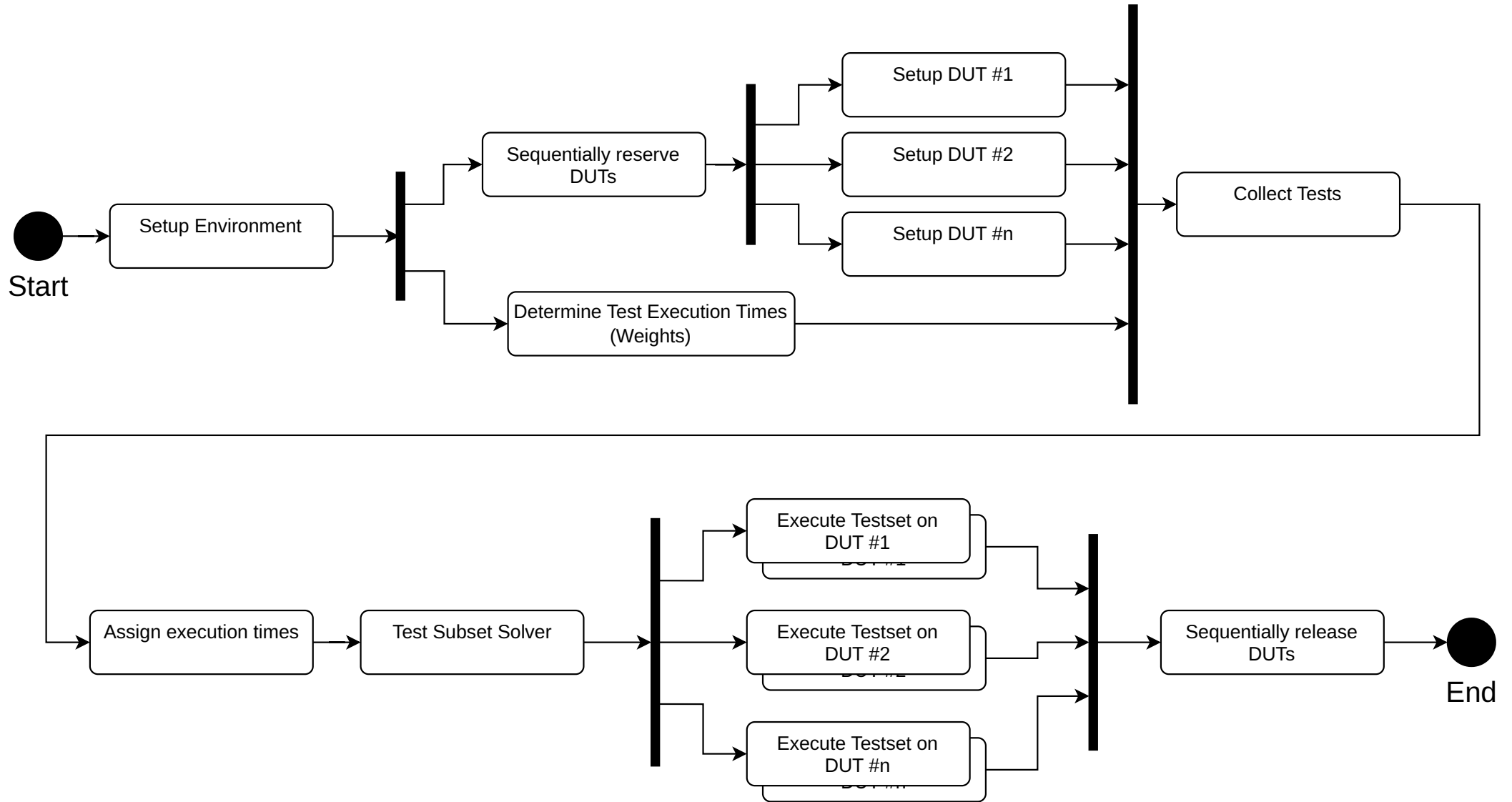
Worker 4



# 1D Cutting Stock Solver (3)



# The Big Picture



# Synchronization Mechanisms

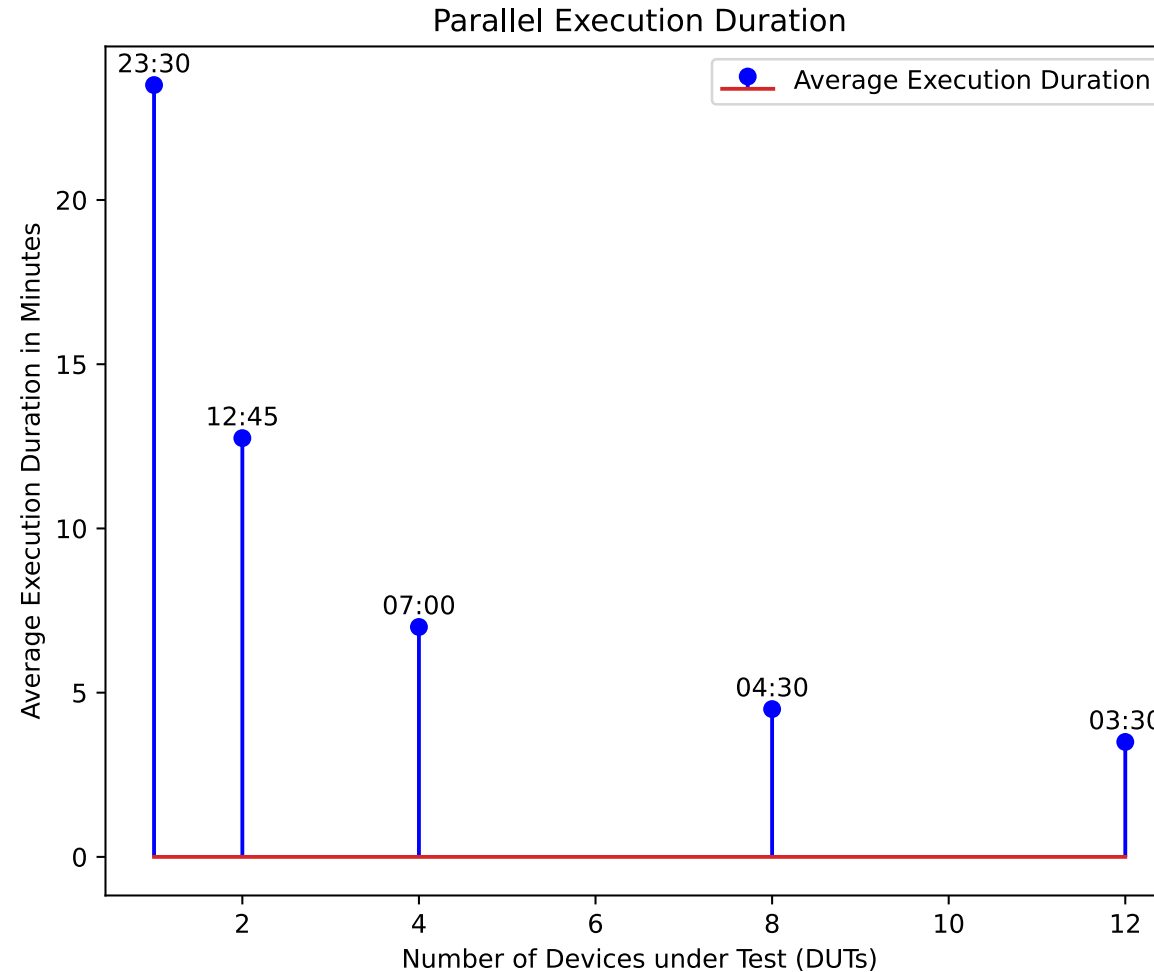
- Shared Resources
- Distributed Locks
  - locks, events, conditions, semaphores
- Introduces complexity across node boundaries
- Dynamic Port ranges
  - predetermined resource locks
  - pytest/labgrid Docker integration

# Parallel Execution Duration (1)

- Doubling the DUT count almost always halves the execution time
- Ceiling is apparent asymptotically

<b>DUTs</b>	<b>Time (min)</b>
1	23:30
2	12:45
4	7:00
8	4:30
12	3:30

# Parallel Execution Duration (2)



## Parallel Execution Duration

# The Importance of Test Case Optimization

- Reaching the apparent ceiling
- Test case optimization is paramount
- Longest single test case == shortest CI/CD runtime
- At any  $n_{max} + 1$  DUTs

# Speedup of Parallel Test Execution

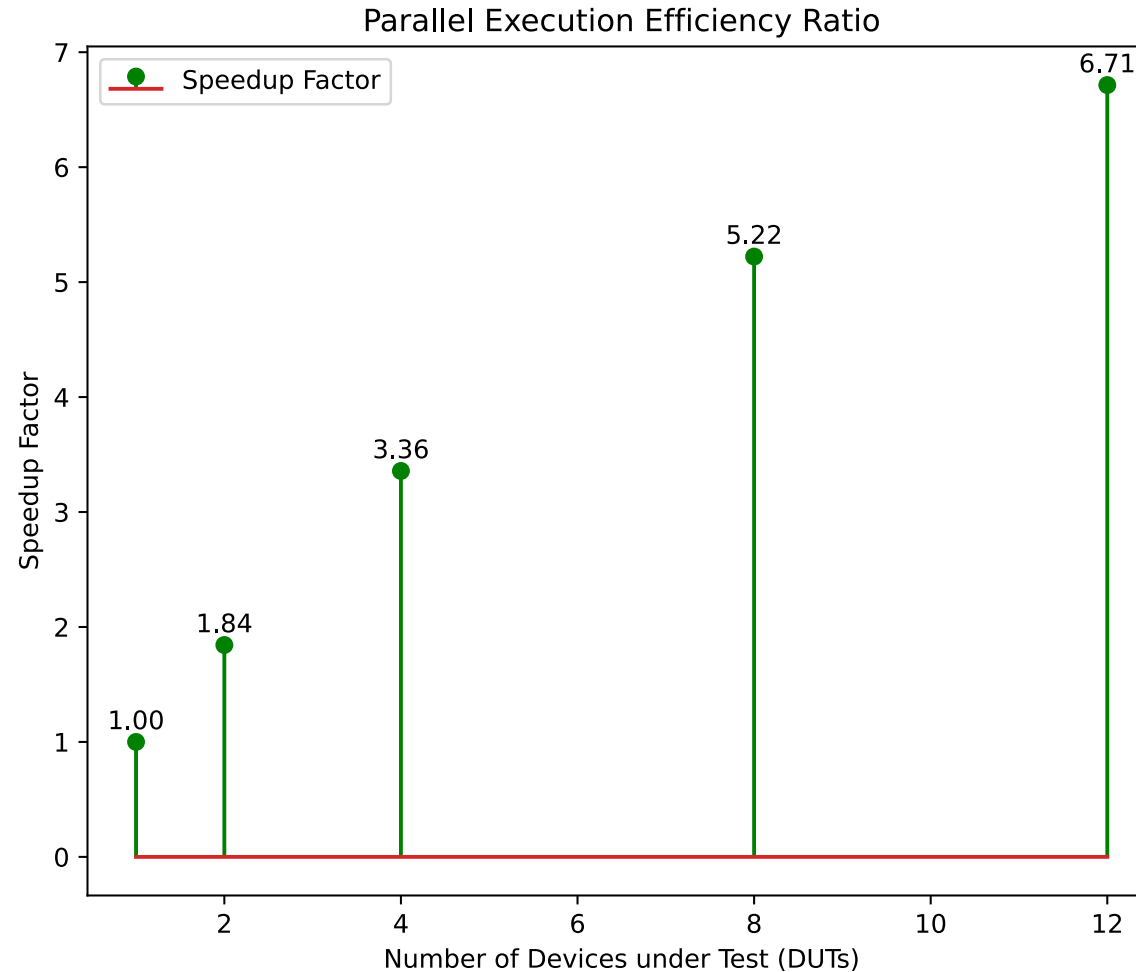
# Detailed Results with 1,600 Tests

- Utilizing this strategy on 1,600 tests
  - unoptimized codebase
    - no weight annotations
    - no special test case separation
- Resulted in a speedup factor of **7.5x**
  - Sequential: 4 h 24 min
  - 20 DUTs: 35 min

# Further case with 110 Tests

- Utilizing this strategy on 110 tests
  - semi-optimized codebase
    - some weight annotations
    - some test case separation
- Resulted in a speedup factor of **6.7x**
  - Sequential: 23 min 30 sec
  - 12 DUTs: 3 min 30 sec

# Parallel Execution Efficiency Ratio



## Parallel Speed Gains

# Calculating the Speedup Ceiling

- Speedup Ceiling depends on
  - accumulated test run duration (sequential)
  - longest running single test

$$\textit{Speedup Ceiling} = \frac{23 \times 60 + 30}{2 \times 60 + 50} \approx 8.30$$

- Accumulated Duration: 23 min 30 sec.
- Longest running test: 2 min 30 sec.
- In comparison 12 DUTs: 3 min 30 sec

# The Need for Effective Test Optimization

- As we approach the speedup ceiling
- Focus shifts to test case optimization
- Reducing single test case runtime
  - Split up into separate tests
  - Optimize logic

# Conclusion and Outlook

# Viabile strategy

- Current
  - Utilizing multiple DUTs, yields a massive gain
  - Ceiling is a factor
  - Manual weight determination
  - Test case optimization from the get go
- Future
  - Dynamic feedback loop for weights
  - Heuristics during discovery phase

# Addendum

# References

## **pytest**

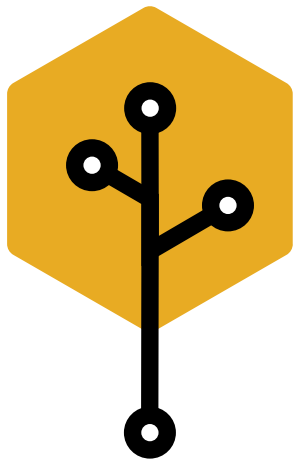
<https://pytest.org>

## **labgrid**

<https://pengutronix.de/en/software/labgrid.html>

# Our Services

- Embedded SW Development
- Continuous Integration
- Continuous Development
- IIoT Security
- RF Protocols
- Reverse Engineering



**honeytree**  
Labs

