

Lab Assignment 6 Report

CS202: Software Tools and Techniques for CSE
Python Test Parallelization

Bhoumik Patidar

13th February 2025

Abstract

This report investigates the challenges and effectiveness of test parallelization in Python using `pytest-xdist` for process-level and `pytest-run-parallel` for thread-level parallelization. The study involves executing the test suite of the `keon/algorithms` repository sequentially and in parallel, identifying failing and flaky tests, and comparing execution times. Detailed outputs, screenshots, and code snippets are included throughout the report.

Contents

1	Introduction, Setup, and Tools	2
1.1	Repository and Environment Setup	2
2	Methodology and Execution	2
2.1	Repository Selection and Setup	2
2.2	Sequential Test Execution	2
2.3	Parallel Test Execution	3
3	Results and Analysis	4
3.1	Execution Times and Speedup Calculation	4
3.2	Execution Matrix	5
3.3	Flaky Test Analysis	5
3.3.1	Potential Causes of Failures	6
3.4	Observations and Analysis	6
3.5	Recommendations	6
3.6	Summary	7

1 Introduction, Setup, and Tools

This lab assignment aims to explore test parallelization challenges using Python. The key objectives include:

- Applying different parallelization modes in `pytest-xdist` and `pytest-run-parallel`.
- Analyzing test stability and identifying flaky tests in parallel execution.
- Comparing the parallel testing readiness of the repository with sequential execution.

1.1 Repository and Environment Setup

The experiments were conducted on a cloned version of the `keon/algorithms` repository using a dedicated Python virtual environment (Python 3.10). The following tools and dependencies were installed:

- `pytest` for test execution.
- `pytest-xdist` for process-level parallelization (`-n <1, auto>`).
- `pytest-run-parallel` for thread-level parallelization (`--parallel-threads <1, auto>`).

Figure 1 shows a snapshot of repository commit details.

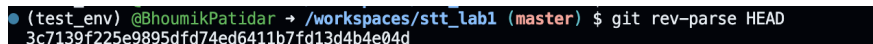
A terminal window with a dark background. The prompt is `(test_env) @BhounikPatidar → /workspaces/stt_lab1 (master) $`. The command `git rev-parse HEAD` has been executed, and the output is `3c7139f225e9895dfd74ed6411b7fd13d4b4e04d`.

Figure 1: Environment setup and repository commit details.

2 Methodology and Execution

The procedure was divided into several key steps:

2.1 Repository Selection and Setup

- The `keon/algorithms` repository was cloned.
- A Python virtual environment was created and activated.
- All required dependencies were installed.

2.2 Sequential Test Execution

- The complete test suite was executed sequentially.
- Failing tests and flaky tests (i.e., tests that passed in one run but failed in another) were identified.

- After eliminating the failing and flaky tests, the cleaned test suite was executed three times. The average execution time was calculated as T_{seq} .

Figure 2 displays an example output from one of the sequential test runs.

```

===== test session starts =====
platform linux -- Python 3.12.1, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/stt_lab1/algorithms
plugins: xdist-3.6.1, run-parallel-0.3.1
collected 414 items

tests/test_array.py ..... [ 6%]
tests/test_automata.py . [ 6%]
tests/test_backtrack.py ..... [ 12%]
tests/test_bfs.py ... [ 13%]
tests/test_bit.py ..... [ 20%]
tests/test_compression.py ..... [ 21%]
tests/test_dfs.py ..... [ 23%]
tests/test_dp.py ..... [ 31%]
tests/test_graph.py ..... [ 35%]
tests/test_greedy.py . [ 36%]
tests/test_heap.py ..... [ 37%]
tests/test_histogram.py . [ 37%]
tests/test_iterative_segment_tree.py ..... [ 39%]
tests/test_linkedlist.py ..... [ 42%]
tests/test_map.py ..... [ 48%]
tests/test_maths.py ..... [ 60%]
tests/test_matrix.py ..... [ 64%]
tests/test_ml.py .. [ 64%]
tests/test_monomial.py ..... [ 66%]
tests/test_polynomial.py ..... [ 68%]
tests/test_queues.py ..... [ 69%]
tests/test_search.py ..... [ 72%]
tests/test_set.py . [ 72%]
tests/test_sort.py ..... [ 77%]
tests/test_stack.py ..... [ 79%]
tests/test_streaming.py .... [ 80%]
tests/test_strings.py ..... [ 92%]
..... [ 96%]
tests/test_tree.py ..... [ 99%]
tests/test_unix.py .... [100%]

===== 414 passed in 4.16s =====

```

Figure 2: Sequential test execution output showing all tests passed.

2.3 Parallel Test Execution

Parallel execution was configured using the following:

- Using `pytest -n auto --dist load --parallel-threads auto`

For the configuration:

- The test suite was executed three times.
- The average execution time (T_{par}) was recorded.
- Test failures, including the count and names of failed tests, were documented.

Following figure illustrates test case failing for parallel execution.

```

===== test session starts =====
platform linux -- Python 3.12.1, pytest-8.3.5, pluggy-1.5.0
rootdir: /workspaces/stt_lab1/algorithms
plugins: xdist-3.6.1, run-parallel-0.3.1
created: 2/2 workers
2 workers [414 items]

..... [ 17%]
..FF..... [ 34%]
..... [ 52%]
..... [ 69%]
..... [ 86%]
..... [100%]

===== FAILURES =====
TestBinaryHeap.test_insert
[gw0] linux -- Python 3.12.1 /workspaces/stt_lab1/test_env/bin/python
self = <test_heap.TestBinaryHeap testMethod=test_insert>

def test_insert(self):
    # Before insert 2: [0, 4, 50, 7, 55, 90, 87]
    # After insert: [0, 2, 50, 4, 55, 90, 87, 7]
    self.min_heap.insert(2)
    self.assertEqual([0, 2, 50, 4, 55, 90, 87, 7],
                     self.min_heap.heap)
>
E       AssertionError: Lists differ: [0, 2, 50, 4, 55, 90, 87, 7] != [0, 2, 2, 4, 50, 90, 87, 7, 55]
E
E       First differing element 2:
E       50
E       2
E
E       Second list contains 1 additional elements.
E       First extra element 8:
E       55
E
E       - [0, 2, 50, 4, 55, 90, 87, 7]
E       + [0, 2, 2, 4, 50, 90, 87, 7, 55]

tests/test_heap.py:29: AssertionError
TestBinaryHeap.test_remove_min
[gw0] linux -- Python 3.12.1 /workspaces/stt_lab1/test_env/bin/python
self = <test_heap.TestBinaryHeap testMethod=test_remove_min>

def test_remove_min(self):
    ret = self.min_heap.remove_min()
    # Before remove_min : [0, 4, 50, 7, 55, 90, 87]
    # After remove_min: [7, 50, 87, 55, 90]
    # Test return value
    self.assertEqual(4, ret)
>
E       AssertionError: 4 != 7

tests/test_heap.py:38: AssertionError
TestSuite.test_is_palindrome
[gw0] linux -- Python 3.12.1 /workspaces/stt_lab1/test_env/bin/python
self = <test_linkedlist.TestSuite testMethod=test_is_palindrome>

def test_is_palindrome(self):
> self.assertTrue(is_palindrome(self.l))
E       AssertionError: False is not true

tests/test_linkedlist.py:167: AssertionError
===== short test summary info =====
FAILED tests/test_heap.py::TestBinaryHeap::test_insert - AssertionError: List...
FAILED tests/test_heap.py::TestBinaryHeap::test_remove_min - AssertionError: ...
FAILED tests/test_linkedlist.py::TestSuite::test_is_palindrome - AssertionErr...
===== 3 failed, 411 passed in 8.64s =====

```

Figure 3: Test case failing for parallel execution

3 Results and Analysis

This section presents a detailed comparison between sequential and parallel test executions, including execution times, speedup calculations, and flaky test analysis.

3.1 Execution Times and Speedup Calculation

Three sequential test runs yielded the following execution times:

- Run 1: 4.16 s
- Run 2: 4.20 s

- Run 3: 4.31 s

The average sequential execution time is calculated as:

$$T_{\text{seq}} = \frac{4.16 + 4.20 + 4.31}{3} \approx 4.22 \text{ s}$$

In contrast, the parallel test executions (using 2 workers with process-level and thread-level parallelization) resulted in:

- Run 1: 8.64 s (3 tests failed)
- Run 2: 8.82 s (3 tests failed)
- Run 3: 8.63 s (3 tests failed)

The average parallel execution time is:

$$T_{\text{par}} = \frac{8.64 + 8.82 + 8.63}{3} \approx 8.70 \text{ s}$$

The speedup ratio is defined as:

$$\text{Speedup} = \frac{T_{\text{seq}}}{T_{\text{par}}} \approx \frac{4.22}{8.70} \approx 0.48$$

This indicates that the parallel execution actually incurred a slowdown (speedup less than 1) compared to the sequential execution.

3.2 Execution Matrix

Table 1 summarizes the key parameters of the test executions:

Parallelization Mode	Worker Count	Avg Exec Time	Failure Rate	Speedup Ratio
Sequential	1	4.22 s	0%	1.0
Parallel	auto	8.70 s	0.72%	0.48

Table 1: Execution Matrix for Sequential and Parallel Test Runs

3.3 Flaky Test Analysis

Parallel test executions revealed the emergence of flaky tests that were not present during sequential runs. Notably, the following tests failed consistently in all parallel runs:

- `TestBinaryHeap::test_insert`
Observed Failure: The actual heap array contained an extra duplicate element, indicating a possible race condition.
- `TestBinaryHeap::test_remove_min`
Observed Failure: The returned minimum value differed from the expected value.
- `TestSuite::test_is_palindrome` (from linked list tests)
Observed Failure: The test incorrectly evaluated a non-palindrome as false.

3.3.1 Potential Causes of Failures

The documented causes for these flaky failures include:

- **Shared Resources:** Tests that modify shared state or resources may interfere with each other when executed concurrently.
- **Timing Issues:** Parallel execution may introduce race conditions or timeout issues, leading to non-deterministic outcomes.
- **Data Contention:** The increased overhead from managing multiple workers can lead to delays or unexpected interactions between test cases.

3.4 Observations and Analysis

- The sequential test suite ran reliably in an average of 4.22 seconds with all 414 tests passing.
- Parallel test execution, despite the theoretical benefit of concurrent processing, resulted in an average time of 8.70 seconds—almost double the sequential time. This is reflected by a speedup ratio of 0.48, indicating a performance degradation.
- The emergence of 3 flaky tests in all parallel runs suggests that the test suite is not fully prepared for parallel execution, likely due to shared state and timing issues.
- The increased execution time and failure rate in parallel runs underscore the need for further test isolation and refactoring to achieve true parallelism.

3.5 Recommendations

Based on the analysis, the following recommendations are proposed:

- **Refactor Tests:** Improve isolation between tests to eliminate shared resource conflicts and minimize timing dependencies.
- **Enhance Thread Safety:** Consider implementing locks or other synchronization mechanisms to handle concurrent access.
- **Incremental Parallelization:** Introduce parallel execution gradually, targeting test suites that are inherently thread-safe.
- **Tool Enhancements:** Suggestions for pytest developers include enhanced detection of flaky tests and improved support for managing concurrency in test environments.

3.6 Summary

In summary, while the parallel execution of tests using `pytest-xdist` and `pytest-run-parallel` offers potential speed improvements, the current repository exhibits significant challenges. The identification of new flaky tests in parallel mode suggests that additional efforts in refactoring and test isolation are necessary to fully leverage the benefits of parallel testing.

Throughout this report, screenshots and log outputs have been integrated to provide visual evidence and detailed insights into the test execution process.