

Timing Sensitivity in the Parsl Test Suite: A Study Based on Controlled Delay Injection

Arthur Prince de Almeida¹, Alfredo Goldman¹

¹ Instituto de Matemática e Estatística
Universidade de São Paulo (USP)
São Paulo, SP, Brasil

{r2.arthur.prince,gold}@ime.usp.br

Abstract. *Testing parallel and distributed software is challenging because test outcomes may depend on scheduling, asynchronous communication, timeouts, and the relative progress of threads and processes. In this paper, we investigate timing sensitivity in the test suite of Parsl, a Python library for parallel and distributed computing. We first evaluate controlled delay injection for exposing timing-sensitive tests by perturbing concurrent execution components and observing the resulting failures. We then analyze the exposed tests to determine the causes of their failures and perform a manual analysis of the complete test suite to identify additional tests whose behavior depends on timing and to characterize the mechanisms used to mitigate nondeterminism, such as condition-based polling, wall-clock constraints, and fixed sleeps. Our study provides a characterization of timing-sensitive tests in a real-world parallel runtime and examines how controlled execution delays affect test outcomes. The results showed that controlled delay injection exposed 7 tests among the 536 test cases evaluated, while manual analysis identified 21 timing-sensitive tests. These findings provide insights into common sources of timing dependency and demonstrate how controlled delay injection can be used to study timing-sensitive tests in parallel and distributed software.*

1. Introduction

Scientific software has become increasingly large and complex, increasing the need for systematic software engineering practices to support its development, maintenance, and long-term evolution [Arvanitou et al. 2021]. This concern is particularly relevant in high-performance computing (HPC), where the adoption of software engineering practices has long been recognized as important for the development of reliable and sustainable scientific software [Basili et al. 2008]. In this context, testing has received increasing attention as a software engineering practice in scientific software [Arvanitou et al. 2021].

Testing concurrent and distributed software is challenging because program behavior may depend on nondeterministic execution ordering, asynchronous operations, and communication delays [Deligiannis et al. 2016]. Such timing dependencies can also affect test code, contributing to nondeterministic test outcomes [Luo et al. 2014]. Techniques that perturb program execution have therefore been investigated as a means of exposing concurrency- and timing-dependent behavior [Křena et al. 2012].

Parsl [Parsl Project] is a Python library for parallel and distributed programming designed to support scientific workflows across computing environments ranging

from local machines to large HPC systems [Babuji et al. 2019]. Its event-driven runtime coordinates task dependencies, asynchronous execution, communication, and resource management across multiple runtime components. Parsl has also demonstrated large-scale execution in HPC environments, with the HighThroughputExecutor (HTEX) scaling to 65,536 workers across 2,048 compute nodes while executing about 640,000 tasks [Babuji et al. 2019]. Its scientific relevance, demonstrated scalability, and asynchronous distributed runtime make Parsl a suitable real-world system for investigating the effects of controlled timing perturbations on both its runtime and test suite.

In this work, we investigate the effects of controlled delay injection on the Parsl test suite and whether such perturbations can expose timing-dependent tests or concurrency and synchronization problems in the runtime. We introduce delays at selected execution points of asynchronous runtime components and execute the existing test suite under these perturbed conditions. Affected tests are then manually analyzed to determine the causes of their failures. We subsequently inspect the complete test suite for explicit timing dependencies to identify cases that may not be exposed by the selected delay injection points. Our experiments identify multiple tests affected by changes in the relative progress of runtime components, including cases in which the failures reveal fragile assumptions in test design rather than incorrect behavior in the Parsl runtime.

2. Related Work

Concurrency bugs may depend on specific execution interleavings that are difficult to exercise through conventional test execution. Noise injection is one approach for increasing the diversity of observed executions by deliberately perturbing the progress of concurrent activities [Křena et al. 2012]. Previous work has used scheduling perturbations to increase the likelihood of manifesting concurrency bugs and subsequently minimized the injected noise to identify the small set of execution points relevant to a failure [Eytani and Latvala 2006]. Other studies have investigated how noise should be generated and where it should be introduced, showing that both the perturbation mechanism and the selection of injection points can substantially affect the behaviors and failures exposed during testing [Křena et al. 2012].

Timing dependencies are also an important source of test flakiness. Asynchronous waits and concurrency have been identified among the most frequent causes of flaky tests, and several such failures can be manifested by modifying or introducing delays [Luo et al. 2014]. Approaches for exposing these failures have perturbed execution in different ways. Shaker adds resource stressors that compete for CPU and memory, changing event ordering and increasing the likelihood of detecting concurrency-related flaky tests [Silva et al. 2020]. FlakeRepro instead combines static and dynamic analysis to identify thread interleavings that reproduce previously observed concurrency-related flaky-test failures [Leesatapornwongsa et al. 2022]. More closely related to our work, FlakeRake inserts delays at timing-dependent API calls and searches for delay configurations that reproduce previously observed timing-dependent flaky-test failures [Rahman et al. 2024].

Our study uses controlled delays for a different objective. Rather than starting from previously observed flaky failures and searching for an execution that reproduces them, we use delay injection to identify tests whose behavior depends on the timing of asynchronous runtime components. The object of study is a parallel-programming library



Figure 1. Simplified Parsl execution flow [Parsl Developers].

in which many tests exercise interactions among threads and processes. We therefore exploit the runtime architecture to select a limited set of asynchronous components as injection points, instead of searching across all potential concurrency locations in the codebase. The behaviors exposed by these perturbations are then analyzed according to their sensitivity to different delay magnitudes. Finally, we complement the automated experiments with a manual analysis to identify additional tests with timing dependencies that were not exposed by the selected delay-injection points.

3. Study Context

3.1. Parsl

Parsl is a research software framework for developing and executing parallel and distributed scientific workflows in Python and is used by researchers and practitioners in high-performance computing (HPC) environments [Babuji et al. 2019]. The idea behind Parsl is to separate application logic from execution infrastructure, allowing the same workflow to run on a local machine, cloud resources, or computing clusters with only small configuration changes.

In Parsl, computational activities are expressed as Python or Bash functions, called *apps*. When an app is invoked, the DataFlowKernel (DFK) determines when it becomes ready for execution, tracks its dependencies, and submits it to an executor. Parsl organizes its execution infrastructure through three main abstractions: executors, providers, and launchers. Executors define how tasks are executed and how their results are returned to the workflow. Providers are responsible for acquiring computational resources, such as execution nodes, CPUs, and memory, while launchers define how Parsl processes are started within those resources. Figure 1 illustrates the relationship between these components.

3.2. Runtime Architecture Relevant to the Study

Although a Parsl application can submit a task through a simple function call, executing that task may involve several asynchronous runtime components operating concurrently. In a minimal HTEX configuration with a single manager and a single worker, Parsl starts at least three additional processes (the interchange, manager, and worker), as well as six background threads distributed across the application and manager processes. When optional services such as monitoring and memoization are enabled, the execution of a single task may involve at least five processes and eight background threads. These numbers

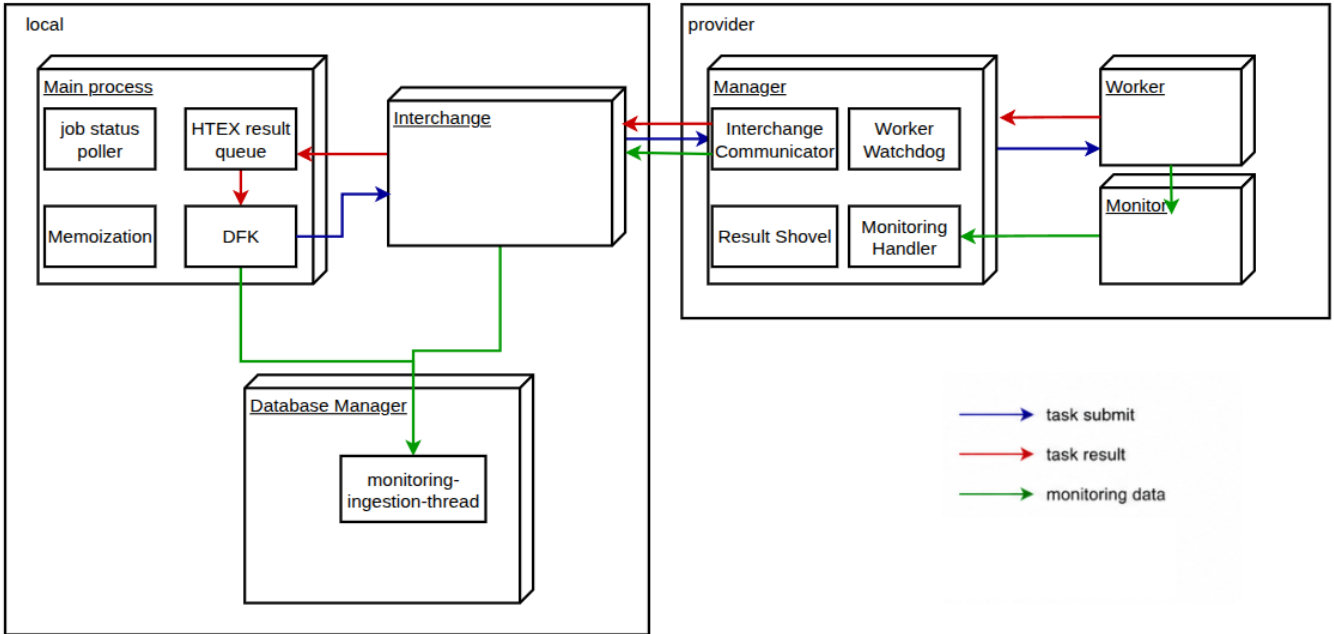


Figure 2. Runtime processes, threads, and communication paths relevant to the study.

correspond to a minimal configuration containing a single resource block and a single worker; larger configurations may create additional managers, workers, and resource-monitor processes. As a result, the execution of a simple application depends on the interaction of multiple concurrent components that communicate through asynchronous channels and progress independently. Consequently, tests that interact with these components may become sensitive to the timing of events and state updates. For this reason, understanding the concurrent runtime architecture of Parsl is important for analyzing the timing dependencies investigated in this study.

Figure 2 presents the HTEx runtime architecture relevant to this study, including its processes, background threads, and communication paths. The DFK and main runtime components execute locally, whereas managers and workers execute on resources allocated through the configured provider.

Main Parsl process. When a configuration containing HTEx is loaded, the executor starts the runtime infrastructure required for task execution, including the interchange process and communication channels using ZeroMQ for asynchronous messaging [Hintjens 2013]. Background components such as the result-queue worker and job-status poller are also started, while memoization is initialized to enable reuse of previously computed results.

When the DFK submits a task, HTEx sends it to the interchange, which dispatches it to a manager when worker capacity is available. In parallel, the job-status poller obtains resource state from the provider, and the scaling strategy may request additional blocks. Each block starts a manager and its workers.

After execution, results travel from the worker through the manager and interchange to the HTEX result queue. The result-queue worker then associates the result or exception with the corresponding DFK future.

Interchange process. The interchange connects HTEX with the managers. It tracks connected managers and available capacity, dispatches tasks, and forwards results and control messages. When monitoring is enabled, it also forwards runtime information. Delays in this process can therefore affect task dispatch, result delivery, monitoring, and manager-state visibility.

Manager process. Each resource block contains a manager that creates and coordinates its workers. Its background threads include the *interchange communicator*, which exchanges tasks and control information; the *result shovel*, which forwards completed results; and the *worker watchdog*, which supervises worker processes. When monitoring is enabled, the *monitoring handler* forwards task-monitoring messages.

Worker and resource-monitor processes. Workers are separate processes that execute tasks received from the manager and return their results or exceptions. When resource monitoring is enabled, a separate monitor process periodically collects CPU, memory, I/O, and other resource-usage metrics.

Database-manager process. The database manager is a separate monitoring process that receives messages from the DFK, interchange, and resource-monitoring components and persists them to the monitoring database. Resource-monitor messages may first pass through an ingestion thread, so monitoring data can remain in transit after the corresponding task has completed.

4. Testing in Parsl

The Parsl test suite combines isolated component tests with tests that start parts of the runtime and exercise communication among concurrent components. A significant portion of the suite is integration-oriented, as tests may instantiate executors, submit apps, and start processes such as the interchange, managers, workers, and monitoring services.

4.1. Test Setup

The Parsl test suite supports two main execution modes. With `-config local`, the suite exercises internal Parsl components, including HTEX, the interchange, and the monitoring infrastructure. These tests inspect internal behavior and are therefore predominantly white-box oriented.

Alternatively, the suite can be executed with a Parsl configuration module:

```
pytest parsl/tests --config path/to/config.py
```

In this mode, the configuration defines the executor, provider, launcher, and other properties of the execution environment. The applicable tests exercise externally observable Parsl functionality, such as task submission, dependency handling, and executor shutdown. They are therefore predominantly black-box and integration oriented.

The two modes do not execute the same set of tests. This separation allows developers and community members to evaluate new executors, providers, or launchers by running the applicable functional tests with a configuration that uses the new component.

This mechanism is also important because the regular continuous-integration (CI) infrastructure cannot reproduce every environment supported by Parsl. In the studied version, Parsl provides 13 providers, while end-to-end provider testing in CI is limited to local resources and Slurm. According to the project maintainers, broader coverage is constrained by the cost and availability of the required computing resources.

4.2. Temporal Coordination in Parsl Tests

Tests that exercise concurrent components must determine when an asynchronous operation has progressed sufficiently for an assertion to be evaluated. Several mechanisms can be used for this purpose, including fixed sleeps, condition-based polling, retries, timeouts, thread or process termination, completed futures, and synchronization events.

One such mechanism is condition-based polling. Instead of immediately asserting that an asynchronous condition is true, a helper repeatedly evaluates the condition until it succeeds or a timeout expires. A simplified version of this pattern is:

```
deadline = current_time() + timeout

while current_time() < deadline:
    if condition():
        return
    sleep(polling_interval)

assert condition()
```

Polling reduces dependence on a single predetermined waiting period. A fast execution can continue as soon as the condition becomes true, while a slower execution is given additional time. It is therefore more flexible than a fixed sleep.

Nevertheless, bounded polling does not eliminate the temporal dependency. It replaces the assumption that an operation completes before one specific observation with the assumption that it completes before a configured deadline. A test may therefore fail when a component is delayed beyond the timeout, even when it would eventually produce the expected state.

Fixed sleeps can also be used when tests exercise explicitly temporal features. These tests wait for a configured duration and then inspect the expected state. Their reliability depends on the difference between the system's configured period and the time allowed by the test before its assertion or timeout.

Explicit synchronization mechanisms provide stronger coordination. Events, joins, completed futures, and other completion signals allow a test to wait for a specific operation rather than estimating its progress from elapsed time. However, such synchronization may not be available for every internal state transition that a test needs to observe.

Thus, these coordination mechanisms can serve two roles. They mitigate non-determinism caused by asynchronous execution, but they also establish temporal bounds

within which the expected behavior must become observable. These bounds, particularly when explicit synchronization is unavailable for some internal states, motivate the controlled-delay study described in Section 5.

5. Methodology

This study combines controlled delay injection with manual analysis of the Parsl test suite. Controlled delays were first introduced at selected execution points to identify tests affected by changes in the relative progress of concurrent components. The resulting failures were then analyzed to determine their causes. Finally, the test suite was manually inspected to identify and classify timing dependencies, including cases not exposed by the selected delay perturbations.

5.1. Study Object and Environment

The study was conducted on the test suite of Parsl version 2026.2.9. The experimental environment used Python 3.13.11, pytest 7.4.4, and the Ubuntu operating system.

Two test scenarios were evaluated. The first used the test suite’s `local` configuration, which exercises internal Parsl components, including HTEX, the interchange, scaling mechanisms, memoization, and monitoring services. The second used `parsl/tests/configs/htex_local.py`, which runs the applicable tests using a locally deployed HighThroughputExecutor (HTEX). This configuration starts the HTEX interchange, managers, and workers without requiring an external batch scheduler. Pytest collected 392 test cases in the `local` scenario and 144 test cases in the `htex_local.py` scenario.

5.2. Controlled Delay Injection

To perturb the relative progress of concurrent Parsl components, we inserted environment-controlled sleep operations at selected execution points. Each injection point was controlled independently through an environment variable associated with both a component and an injection position. The supported positions corresponded to component initialization, the period immediately after its creation, recurring loop iterations, and termination. The value assigned to each variable represented the injected delay in seconds.

The experiments used delays of 0.5, 3, and 5 seconds. These values were selected pragmatically rather than derived from a formal timing model. They provide increasing perturbation magnitudes without introducing excessively long delays into the test executions. Rahman et al. [Rahman et al. 2024] also used a 5-second delay in one of the scenarios evaluated in their study.

The injection positions represent different points at which the progress of a concurrent component or its creator can be perturbed.

An `INIT` delay executes inside a thread or process before its regular work begins. It postpones the initial progress of the component without delaying the code that created it.

A `POST_START` delay executes in the creator immediately after a child thread or process has been started. Unlike an `INIT` delay, it does not prevent the newly created component from progressing. Instead, it delays the creator while the background

component continues to execute. This position can therefore expose assumptions that a background component will not advance beyond an intermediate state before the main execution path inspects it.

A `LOOP` delay executes whenever a selected recurring loop point is reached. It affects components that periodically poll state, process messages, update runtime information, or perform other recurring activities.

An `END` delay executes in the termination path before the thread, process, or background activity finishes. It postpones the completion or shutdown of the selected component.

Not every component supports all four positions. Some components do not contain a long-running loop, while others are created or terminated through control paths for which only a subset of the positions is applicable.

In addition to the component lifecycle positions, the instrumentation included `PARSL_SLEEP_POST_LOAD_DFK`. This delay executes in the main test thread after the `DataFlowKernel` has completed its loading procedure. Its purpose is to pause test execution while allowing the Parsl runtime components to complete initialization, registration, and initial state propagation before the test proceeds.

The instrumentation covered the following runtime activities:

- DataFlowKernel loading;
- Job Status Poller;
- Memoization;
- Interchange Communicator;
- Result Shovel;
- Worker Watchdog;
- Monitoring Handler;
- HTEX Manager;
- HTEX Worker;
- HTEX Interchange;
- HTEX Result Queue Worker;
- Database Manager starter and main loop;
- monitoring resource-message migration; and
- Resource Monitor.

The evaluated variables included initialization, post-start, recurring loop, and termination points where these positions were available. The manager and result-queue instrumentation also included post-start delays, which pause their respective creators after the background activity has begun.

All selected injection points were evaluated regardless of whether they subsequently exposed a test failure. Injection points that produced no failed or errored tests therefore remained part of the experimental space.

5.3. Experimental Procedure

The experiment evaluated 52 sleep-control variables at three delay magnitudes: 0.5, 3, and 5 seconds. Each variable was evaluated independently in both test configurations, with only one injection variable active at a time to avoid interactions between multiple perturbations. For each experimental combination, we applied the following procedure:

1. reset all sleep-control environment variables to zero;
2. assign one of the values 0.5, 3, or 5 seconds to a single injection variable;
3. execute the selected test configuration with a two-minute timeout for each test;

4. record all tests reported by pytest as failed or errored;
5. inspect each reported failure or error to determine whether it was caused solely by the test timeout; and
6. reset the active variable before evaluating the next experimental combination.

For each execution, we recorded the pytest identifiers of all failed or errored tests, together with the active injection point, delay value, and test configuration.

5.4. Failure Analysis

Each test reported as failed or errored under an active delay was manually inspected to determine the cause of the failure and its relationship with the injected perturbation. Failures and errors caused solely by the two-minute experimental timeout were excluded from the analysis.

A test could be exposed by multiple injection points or delay values, but was counted only once in the aggregate results. A test was considered exposed when it failed or errored under an active delay for a reason other than the imposed timeout. Such a failure was interpreted as evidence that the test was sensitive to changes in the relative progress of the perturbed component.

5.5. Manual Analysis

Following the analysis of the delay-exposed failures, we manually inspected the Parsl test suite to identify tests whose behavior depended on elapsed time or on the relative progress of concurrent components. The analysis considered explicit temporal operations, such as sleeps, timeouts, timestamp comparisons, and periodic execution, as well as implicit assumptions about asynchronous state transitions, scheduling, communication, and persistence. When the temporal dependency was not clear from the test alone, we inspected the corresponding Parsl implementation to determine which component or asynchronous behavior the test was observing.

For each identified test, we recorded its pytest identifier, subsystem, temporal mechanism, and the asynchronous behavior involved. Tests were classified according to the predominant mechanism:

1. *bounded polling*, in which a condition is repeatedly evaluated until it becomes true or a timeout expires;
2. *wall-clock assertions*, in which the expected behavior must occur within a specified time interval;
3. *fixed sleeps*, in which the test waits for a predetermined duration before inspecting a state; and
4. *concurrent scheduling assumptions*, in which the result depends on the ordering or relative progress of concurrent activities.

We also examined whether each temporal mechanism was necessary for the behavior being tested. Tests that were not executed or analyzed were kept separate from the manually classified set.

6. Results

6.1. Delay Injection

After excluding failures caused only by the external timeout of the experimental runner, delay injection exposed seven distinct tests. All failures occurred in the `local` scenario. The number of exposed tests increased with the magnitude of the injected delay. One test failed with a delay of 0.5 seconds, four tests failed with 3 seconds, and seven tests failed with 5 seconds. The sets were nested: the test exposed at 0.5 seconds was also exposed at 3 and 5 seconds, and all tests exposed at 3 seconds were also exposed at 5 seconds.

Among the seven exposed tests, one checked whether periodic checkpoints occurred within a short wall-clock interval. It was the most sensitive case, failing with delays as small as 0.5 seconds and at several injection points.

Four exposed tests explicitly waited for asynchronous conditions to become true within bounded intervals. In these cases, the injected delays postponed background operations beyond the temporal bounds used by the tests.

Another test assumed that the scaling thread would progress slowly enough for an intermediate state to remain observable. When the thread advanced faster, the assertion became timing-dependent and the test could fail intermittently.

The remaining exposed test revealed a test-design bug involving concurrent scale-in and scale-out operations. The injected delay allowed the two operations to overlap in different threads, leaving a block marked as available even though it could no longer receive tasks and causing a persistent no-progress state. Inspection of the corresponding test and implementation confirmed that this behavior resulted from the concurrent operations performed by the test.

Table 1. Instrumented activities and exposed behaviors.

Activity	Tests	Main exposed behavior
Job Status Poller	4	Bounded polling and wall-clock assumptions involving scaling and manager-state transitions.
Memoization	1	Sensitivity of periodic checkpoint validation to delays in recurring background processing.
DFK load	3	Timing assumptions affected by allowing background components to progress before test execution continued.
Manager	1	Concurrent scale-in and scale-out producing a persistent no-progress state.
HTEX Result Queue	1	Result delivery exceeding a fixed timeout.

Table 1 relates the instrumented activities to the failures they exposed. Only five of the 14 instrumented runtime activities exposed test failures. Counts overlap because the same test could be exposed by more than one activity.

6.2. Manual Analysis

The manual analysis covered 392 test cases collected with the `local` configuration and 144 test cases collected in the black-box scenario. It identified 21 tests containing timing

dependencies, all of them in the `local` scenario. No timing dependencies were identified among the 144 black-box tests. All seven tests previously exposed through delay injection were included among these 21 tests.

Table 2 groups the identified tests by subsystem and by the predominant mechanism responsible for the dependency.

Table 2. Timing dependencies identified through manual analysis.

Subsystem	Tests	Main mechanisms
Scaling	5	5 polling
Monitoring	4	2 polling, 2 fixed sleeps
HTEX internals	6	5 polling, 1 concurrency
Memoization	2	1 polling, 1 wall-clock
Application execution	4	1 polling, 2 wall-clock, 1 concurrency
Total	21	

The analysis also identified two temporal mechanisms that were unnecessary for the behavior being tested: a fixed sleep and a bounded polling loop applied to a deterministic condition.

7. Discussion

The manual analysis identified 21 timing-sensitive tests among the 392 test cases collected in the local scenario. None were identified among the 144 test cases collected in the black-box scenario. Bounded polling was the most frequent mechanism, followed by wall-clock assertions, fixed sleeps, and assumptions about concurrent execution. Two tests also used temporal barriers that were unnecessary for the behavior under test.

No timing-sensitive tests were identified among the black-box tests. These tests primarily validate externally observable functionality, such as task execution and result delivery, and therefore depend less on the timing of internal states and lifecycle transitions. The failures were also concentrated in a small part of the instrumented runtime: only 5 of the 14 instrumented runtime activities exposed any test failure. The concentration of injected failures in memoization and the Job Status Poller is consistent with their periodic execution, since perturbing these components directly affected checkpointing, scaling decisions, and state updates, thereby exposing tests based on wall-clock limits or bounded polling.

Two failures instead revealed problems in test design. In one case, delaying the completion of manager-launch bookkeeping allowed scale-in and scale-out to overlap in different threads. This left unusable capacity recorded as available and prevented further progress. In the other case, delaying the test after system initialization allowed the scaling strategy to progress before an intermediate state was inspected. The assertion therefore depended on the strategy being slow enough for that state to remain observable. Unlike bounded-polling failures, this test failed because the background component progressed too quickly.

These results highlight the difficulty of designing tests for concurrent systems. Temporal mechanisms are often required to observe asynchronous behavior, but they may

also be unnecessary, incorrectly placed, or based on assumptions that progress will occur neither too slowly nor too quickly. Delay injection was therefore useful both for exposing fragile timing assumptions and for revealing concurrency problems in the tests themselves.

8. Threats to Validity

The manual classification of timing-sensitive tests involves some degree of judgment, as it depends on the interpretation of test behavior and the corresponding runtime implementation. To mitigate this threat, we inspected the relevant Parsl components whenever the temporal dependency was not clear from the test alone. The delay-injection methodology is also limited to the selected injection points and delay values, and different components, locations, or delay durations could expose additional timing-sensitive tests. Finally, the absence of failures under a particular perturbation should not be interpreted as evidence that a test has no timing dependency, but only that the evaluated perturbations did not expose one.

9. Conclusion

This paper investigated timing sensitivity in the Parsl test suite through a combination of manual analysis and controlled delay injection. The manual analysis identified 21 timing-sensitive tests, while delay injection exposed seven of them by perturbing the relative progress of selected runtime components. Bounded polling was the predominant temporal mechanism, and the observed failures were concentrated in components with periodic background execution. The experiments also revealed flaws in test design, showing that controlled execution perturbations can expose both timing-sensitive behavior and fragile assumptions about concurrent execution. Overall, these results characterize timing dependencies in the Parsl test suite and show how controlled delay injection can help reveal weaknesses in tests of concurrent and distributed software.

The identified timing dependencies also represent a practical concern for the Parsl test suite, as such dependencies can contribute to flaky behavior and make test outcomes more sensitive to execution conditions. Building on these findings, we intend to investigate systematic approaches for refactoring timing-sensitive tests and evaluating how these dependencies can be reduced without compromising the behaviors the tests are intended to validate. This investigation can provide a basis for improving the affected Parsl tests and contributing the resulting improvements back to the project.

References

- Arvanitou, E.-M., Ampatzoglou, A., Chatzigeorgiou, A., and Carver, J. C. (2021). Software engineering practices for scientific software development: A systematic mapping study. *Journal of Systems and Software*, 172:110848.
- Babuji, Y., Woodard, A., Li, Z., Katz, D. S., Clifford, B., Kumar, R., Lacinski, L., Chard, R., Wozniak, J., Foster, I., Wilde, M., and Chard, K. (2019). Parsl: Pervasive parallel programming in python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.
- Basili, V. R., Carver, J. C., Cruzes, D., Hochstein, L. M., Hollingsworth, J. K., Shull, F., and Zelkowitz, M. V. (2008). Understanding the high-performance-computing community: A software engineer’s perspective. *IEEE Software*, 25(4):29–36.

- Deligiannis, P., McCutchen, M., Thomson, P., Chen, S., Donaldson, A. F., Erickson, J., Huang, C., Lal, A., Mudduluru, R., Qadeer, S., and Schulte, W. (2016). Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, page 249–262, USA. USENIX Association.
- Eytani, Y. and Latvala, T. (2006). Explaining intermittent concurrent bugs by minimizing scheduling noise. volume 4383, pages 183–197.
- Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.
- Křena, B., Letko, Z., and Vojnar, T. (2012). Noise injection heuristics for concurrency testing. In Kotásek, Z., Bouda, J., Černá, I., Sekanina, L., Vojnar, T., and Antoš, D., editors, *Mathematical and Engineering Methods in Computer Science*, pages 123–135, Berlin, Heidelberg. Springer Berlin Heidelberg.
- Leesatapornwongsa, T., Ren, X., and Nath, S. (2022). Flakerepro: automated and efficient reproduction of concurrency-related flaky tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 1509–1520, New York, NY, USA. Association for Computing Machinery.
- Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 643–653, New York, NY, USA. Association for Computing Machinery.
- Parsl Developers. Quickstart. <https://parsl.readthedocs.io/en/stable/quickstart.html#execution-providers>. Accessed: June 2026.
- Parsl Project. Parsl: Parallel scripting in python. <https://parsl-project.org/>. Accessed: 2026-08-06.
- Rahman, S., Massey, A., Lam, W., Shi, A., and Bell, J. (2024). Automatically reproducing timing-dependent flaky-test failures. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 269–280.
- Silva, D., Teixeira, L., and d'Amorim, M. (2020). Shake it! detecting flaky tests caused by concurrency with shaker. pages 301–311.