Parallel Bank Transactions & Fraud Flags

Threads vs Processes — Race Conditions and Synchronization

Adam El Mouddene (ID: 2023006464)
Amir Beshir (ID: 2023006017)
Hazim Kaloub (ID: 2023005883)
Riyad Almasri (ID: 2023006329)
Sulaiman Alchami (ID: 2023006185)

CSCI 415 — Introduction to Parallel Programming
American University of Ras Al Khaimah
Spring 2026

1 Introduction

This project implements a simplified bank transaction processing system that executes deposits, withdrawals, and transfers on shared account balances. The system is built in Python with a Streamlit dashboard and supports five execution modes: a serial baseline, parallel threads without locks, parallel threads with locks, parallel processes without locks, and parallel processes with locks. The goal is to demonstrate how concurrent access to shared state causes race conditions, show how synchronization primitives fix them, and compare performance and correctness across all modes using standard parallel computing metrics.

2 Race Condition Explanation

A race condition occurs when multiple threads or processes perform a **read–modify–write** sequence on shared data without synchronization. In our system, account balances are stored in a shared dictionary. When two workers process transactions on the same account concurrently, the following interleaving can occur:

1. Worker A reads `balance[X] = 1000`.
2. Worker B reads `balance[X] = 1000` (same stale value).
3. Worker A writes `balance[X] = 1000 - 200 = 800`.
4. Worker B writes `balance[X] = 1000 - 300 = 700`.

The correct result after both withdrawals should be `500`, but the final balance is `700` because Worker B overwrote Worker A's update with a value computed from stale data. This is the classic lost-update problem.

In our implementation, the no-lock modes intentionally use a non-atomic `_racy_write_delta` function that separates the read and write steps with an optional small delay (`race_delay_max`) to increase the probability of interleaving. This makes the race condition observable even with a modest number of transactions.

The effect is measurable: after processing all transactions in no-lock mode, final account balances diverge from the serial baseline. We report this as the number of mismatched accounts and the total absolute error. Transfers are especially prone to races because they involve two balance updates (debit source, credit destination) that must happen atomically.

3 Synchronization Solution

We fix the race conditions by wrapping each transaction's balance reads and writes inside a critical section using locks:

- **Threads:** A single `threading.Lock` guards all balance and ledger mutations. Each call to `_apply_transaction` acquires the lock before reading any balance and releases it only after all writes (including ledger append) are complete. For transfers, both the debit and credit occur within the same `with lock_obj:` block, guaranteeing atomicity.
- **Processes:** Because processes have separate address spaces, shared state is managed through `multiprocessing.Manager()`, which provides proxy objects (`manager.dict`, `manager.list`) accessible across processes. A `manager.Lock()` serves the same role as `threading.Lock` but operates over IPC.

Both modes also use **bounded queues** (`queue.Queue` for threads, `multiprocessing.Queue` for processes) with a configurable `maxsize` to distribute transactions to workers. Graceful shutdown is handled via poison-pill sentinels (`None`) and a `threading.Event` / `multiprocessing.Event` stop flag.

With locks enabled, the locked modes produce final balances that match the serial baseline exactly (zero mismatched accounts, zero total absolute error).

4 Results and Analysis

4.1 Correctness

Table 1 summarizes correctness across modes. No-lock modes show non-zero mismatches and absolute error, confirming the race condition. Locked modes and the serial baseline show perfect agreement.

Table 1: Correctness comparison across modes (example with $p = 4$ workers).

Mode	Mismatched Accounts	Total Abs Error
Serial baseline	0	0.00
Threads, no lock	50	19 633.00
Threads, locked	0	0.00
Processes, no lock	50	17 698.00
Processes, locked	0	0.00

4.2 Performance

Table 2 presents the measured runtime, speedup ($S_p = T_{\text{serial}}/T_{\text{parallel}}$), and efficiency ($E_p = S_p/p$) for every mode and worker count. The serial baseline completed in approximately 0.056 s.

Table 2: Timing performance across all modes and worker counts.

Mode	Workers (p)	Runtime (s)	Speedup S_p	Efficiency E_p
Serial baseline	1	0.056	—	—
Threads, no lock	1	9.733	0.005	0.005
	2	4.726	0.010	0.005
	4	2.381	0.019	0.005
	8	1.222	0.038	0.005
Threads, locked	1	0.102	0.455	0.455
	2	0.106	0.437	0.218
	4	0.084	0.550	0.138
	8	0.101	0.460	0.057
Processes, no lock	1	19.668	0.002	0.002
	2	9.543	0.005	0.002
	4	5.731	0.008	0.002
	8	5.080	0.009	0.001
Processes, locked	1	7.742	0.006	0.006
	2	6.571	0.007	0.004
	4	6.772	0.007	0.002
	8	7.891	0.006	0.001

No parallel mode achieves speedup greater than 1. This is expected for three reasons:

1. **Python's GIL** prevents threads from executing Python bytecode in true parallel, so threaded modes gain no CPU parallelism.

2. **Manager proxy overhead:** every dictionary read/write in process mode crosses a process boundary via IPC, which is orders of magnitude slower than a local dictionary access.

3. **Fine-grained workload:** each transaction is a few arithmetic operations; the coordination overhead (queue put/get, lock acquire/release) far exceeds the computation per task.

Among the parallel modes, **Threads locked** is the fastest (around 0.08–0.10 s), since lock-protected in-process dictionary access avoids IPC entirely. The no-lock thread mode is slower due to the intentional `race_delay_max` sleep injected to expose race conditions. Process modes are the slowest because every shared-state operation passes through a Manager proxy over IPC. Efficiency drops as workers increase for all modes, confirming that coordination overhead dominates the fine-grained per-transaction computation.

These results reflect Python's parallel execution model and are a valuable demonstration that parallelism is not automatically faster.

4.3 Fraud Detection

Two rules are applied to the executed ledger: (1) transactions with amount above a configurable threshold are flagged, and (2) accounts with repeated outflow activity (withdrawals or transfers) within a sliding transaction-ID window are flagged. Fraud flag counts are consistent across locked modes and the serial baseline.

5 Plots and Screenshots

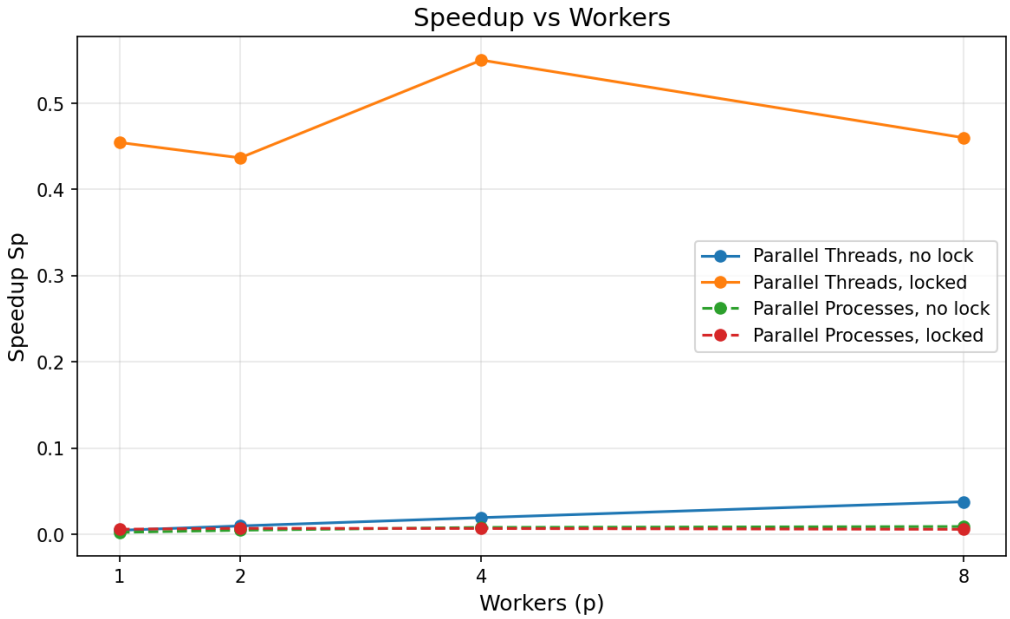

Figure 1: Speedup $S_p = T_{\text{serial}}/T_{\text{parallel}}$ for each mode across $p = 1, 2, 4, 8$ workers.

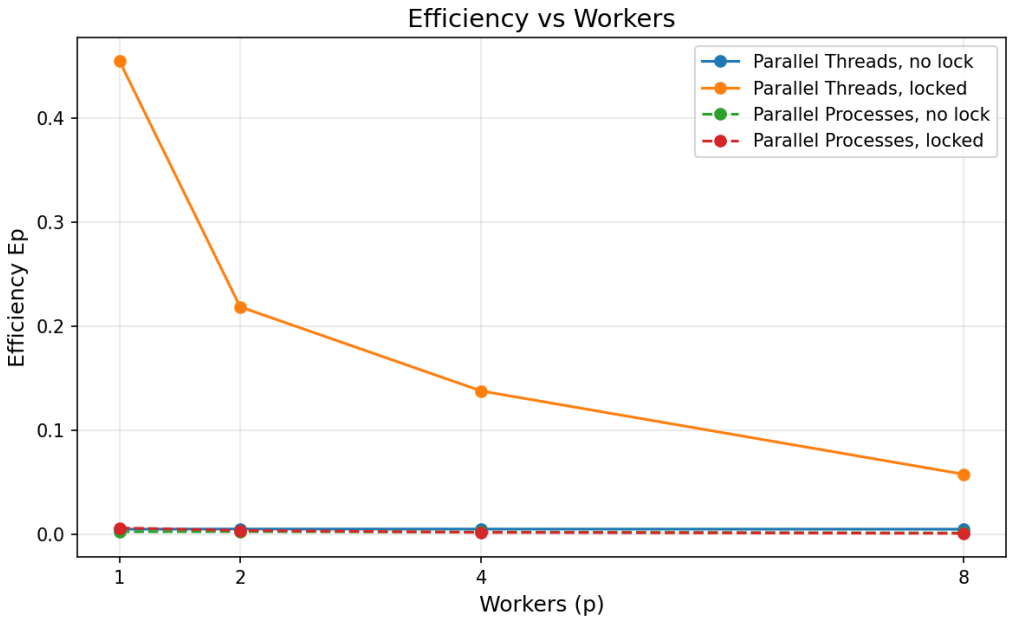

Figure 2: Efficiency $E_p = S_p/p$ for each mode across $p = 1, 2, 4, 8$ workers.

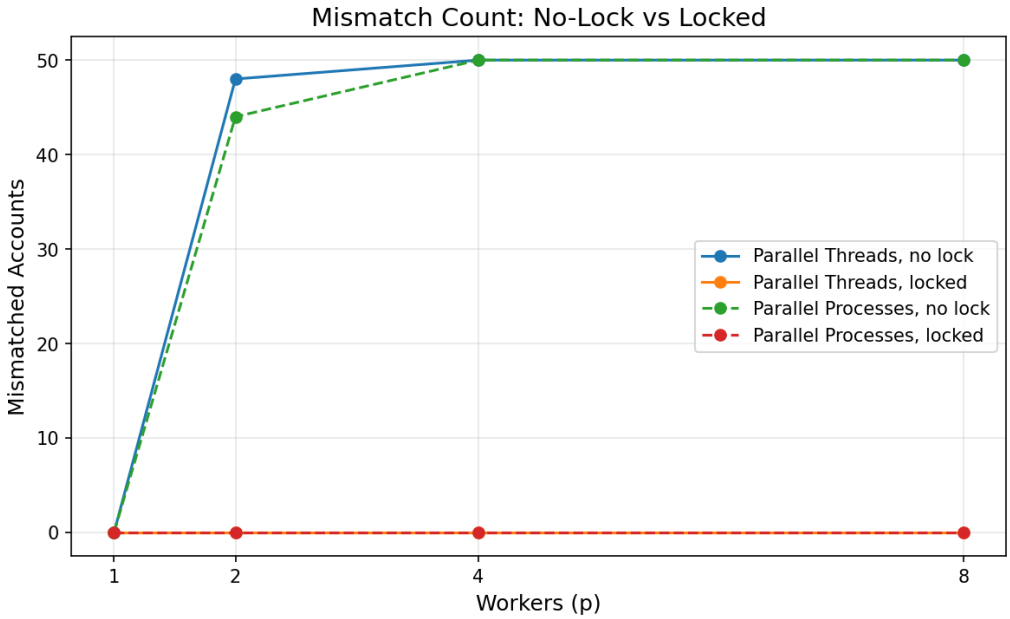

Figure 3: Mismatched account count: no-lock modes show increasing errors with more workers, while locked modes remain at zero.

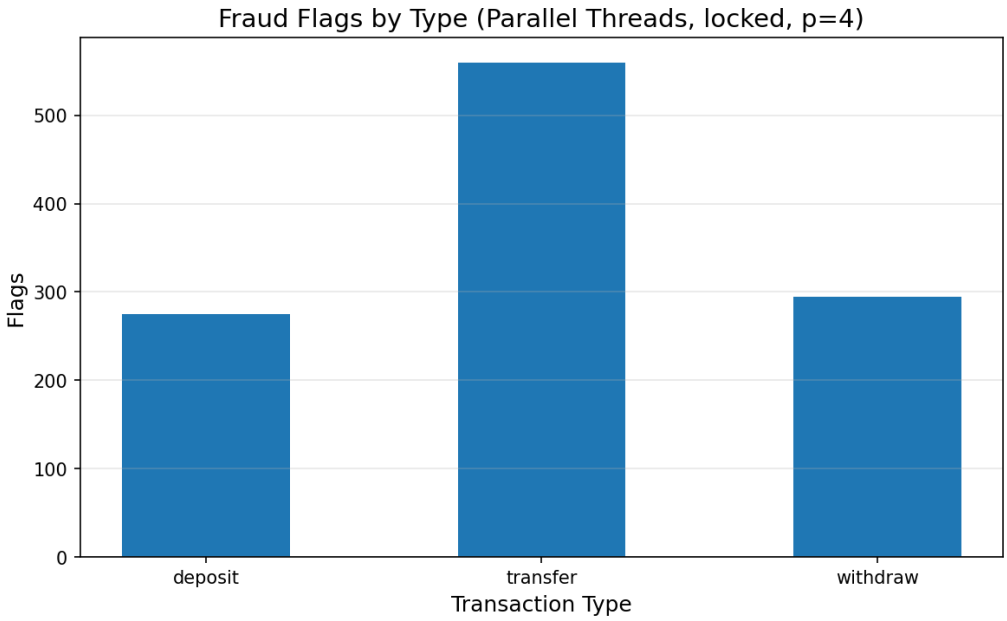

Figure 4: Distribution of fraud flags across transaction types.

6 Conclusion

This project demonstrates that unsynchronized parallel access to shared mutable state leads to incorrect results due to race conditions. Locks eliminate these errors at the cost of serializing critical sections. In Python, thread-based parallelism is limited by the GIL and process-based parallelism incurs significant IPC overhead via Manager proxies, resulting in parallel runtimes slower than the serial baseline for this fine-grained workload. The project successfully fulfills all functional requirements: five execution modes, bounded-queue communication, fraud detection, and comprehensive correctness and performance reporting through an interactive Streamlit dashboard.