Parallel Programming w/ LLMs

Alec, Emir, Julian, Samir, Wil

The Rule

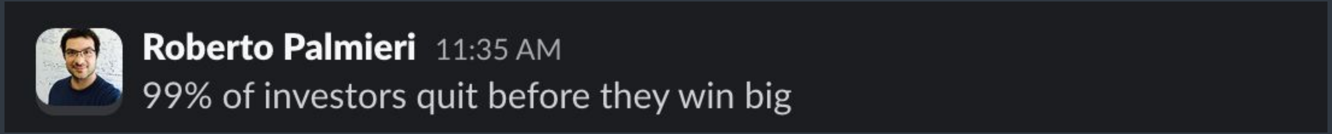
1) No specific instructions

E.g. We can not tell it to implement striped locking for bank

Same prompt != Same result

You are gambling with the odds of getting a good output, but always remember these great words from Professor Palmieri:

“



Roberto Palmieri 11:35 AM

99% of investors quit before they win big

”

- Emir Veziroglu

Starter Prompt

Prompt:

Given this assignment I need you to implement this with a fast implementation. Focus on speeding up the part that is taking the most time. Be sure to maintain the correctness of the code.

Insert assignment instructions here (remove anything that LLM won’t understand e.g. Sunlab, Evaluation, etc.)

Student Prompt

Prompt:

You're an expert in parallel computing with extensive knowledge in developing safe and high-performance parallel code. I need help with a parallel computing homework problem. I've included some background materials and a problem statement below. Optimize it to the best of your ability, using locks (shared or otherwise), condition variables, and/or any other parallel optimization techniques that can improve speed. Please review all the provided context and generate a complete solution for this problem description:

Insert textbook text OR screenshots of relevant textbook pages here.

Insert assignment instructions here (remove anything that LLM won't understand e.g. Sunlab, Evaluation, etc.)

Emir’s Prompt

Prompt:

Implement the project listed. Please parallelize the code you output to maximize speedup on multi-core CPUs. Ensure that parallelization reduces execution time as threads increase, by avoiding shared resource contention, false sharing, and unnecessary synchronization. Identify parts of the code that are free of data dependencies and prioritize them for parallelism. **Suggest alternatives or redesigns for parts that do have dependencies to make them more parallelizable. Optionally use SIMD or memory-friendly access patterns for high-performance computing.**

Keep in mind whenever I ask LLM to generate code for the project below, they alway generate code that does not scale well. **As the number threads increase, so does the execution time. How would you implement this code so that as you increase the number of threads, the execution time goes down.**

Insert assignment instructions here (remove anything that LLM won’t understand e.g. Sunlab, Evaluation, etc.)

That’s not as fast as it can be

Palmieri Prompt aka Beast mode

Prompt:

Imagine you are Professor Palmieri. You are in the top Computer Science and Engineering Department in the world where you lead a research group. **Your research interests focus on concurrency, synchronization, data structures, distributed computing, heterogeneous systems, key-value stores, and distributed systems, spanning from theory to practice. You are passionate about designing and implementing synchronization protocols optimized for a wide range of deployments, from multicore architectures to cluster-scale and geo-distributed infrastructures. How would you implement the following project using research level coding?** I want to know your thoughts on implementation. **Don't implement it, tell me what you would do specifically to speed up this execution multi-threaded part of this code.** BE EXTREMELY CREATIVE. LIST MORE THAN ONE WAY. Come up with a strategy to do the following assignment:

Insert assignment instructions here (remove anything that LLM won't understand e.g. Sunlab, Evaluation, etc.)

```
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
XX
```

Bank

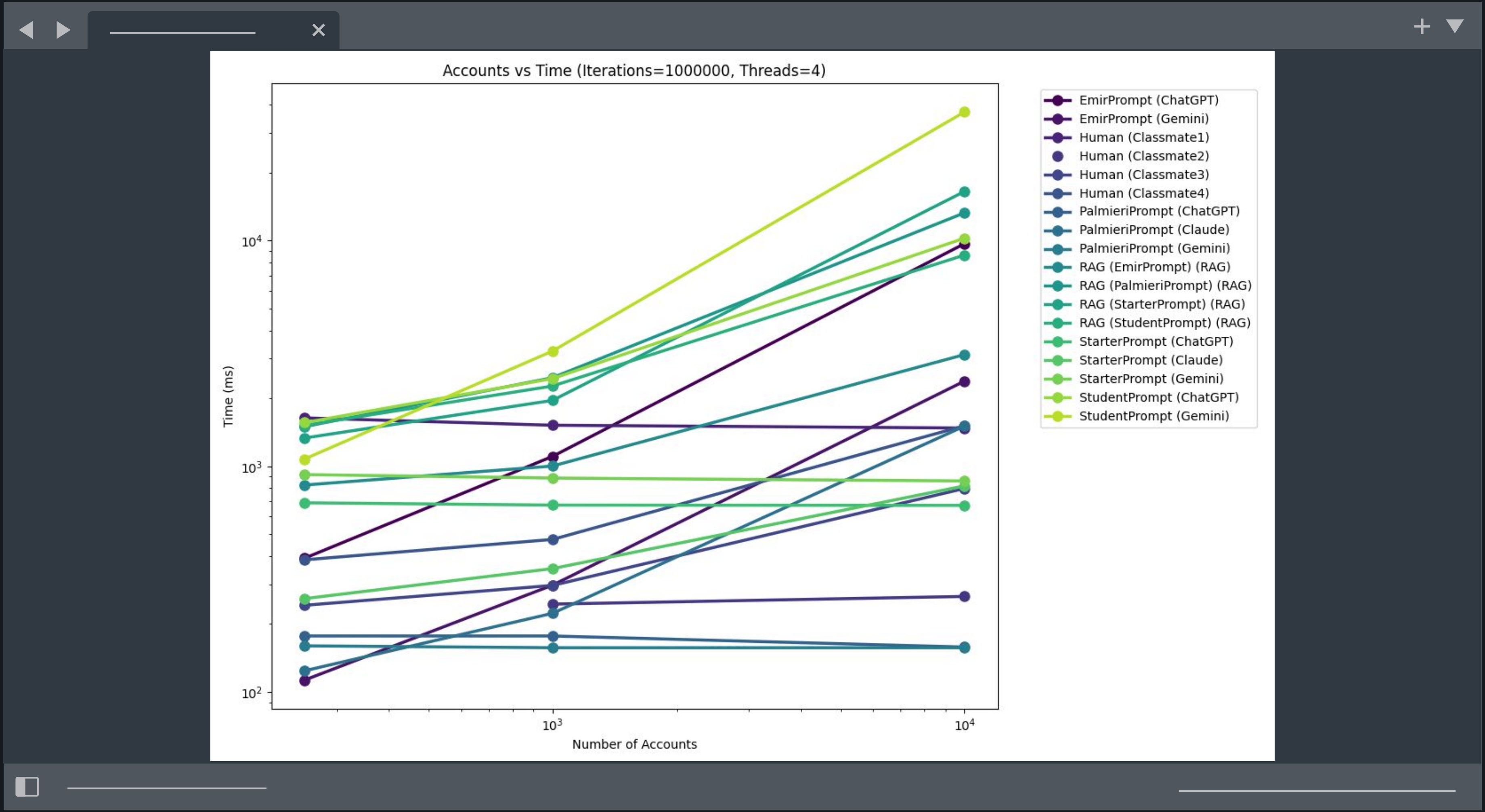
Quick Benchmarking

Task: Bank account update loop (sequential baseline vs. multi-threaded)

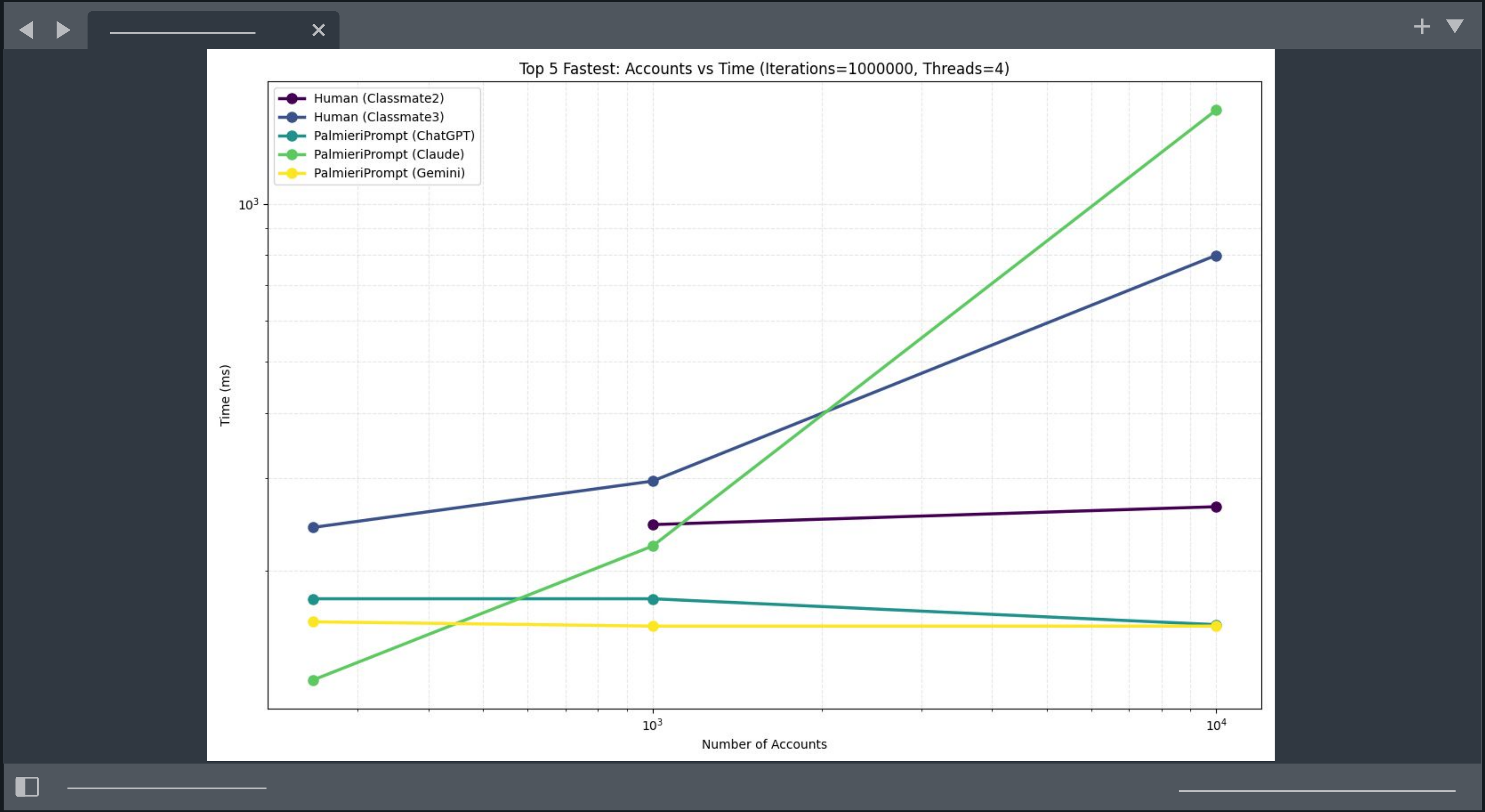
- Prompts to first “good” response (ignoring clarifying questions)
- “Fix this” iterations to pass compilation/tests
- Max thread execution time (not average)

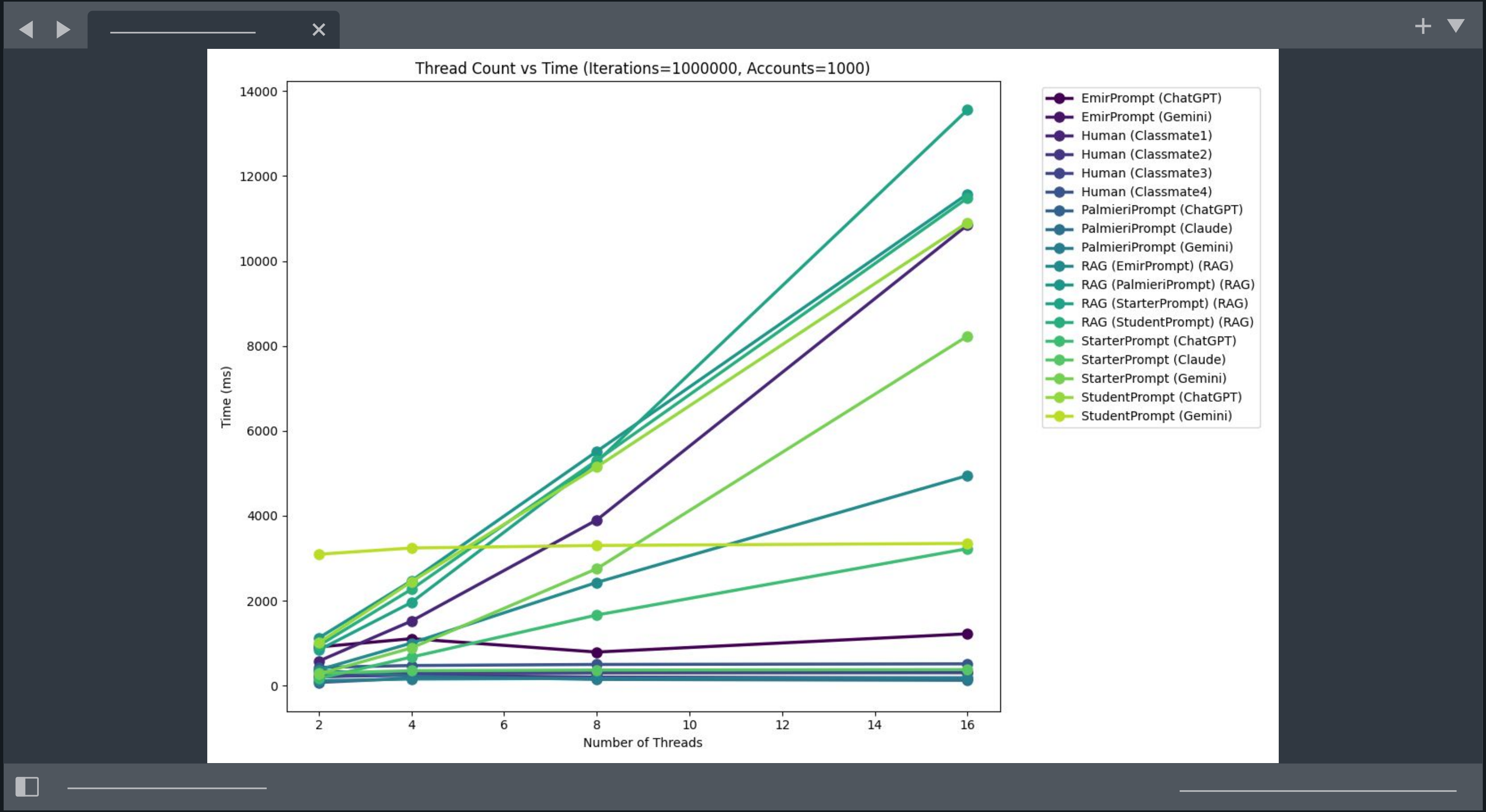
Parameters:

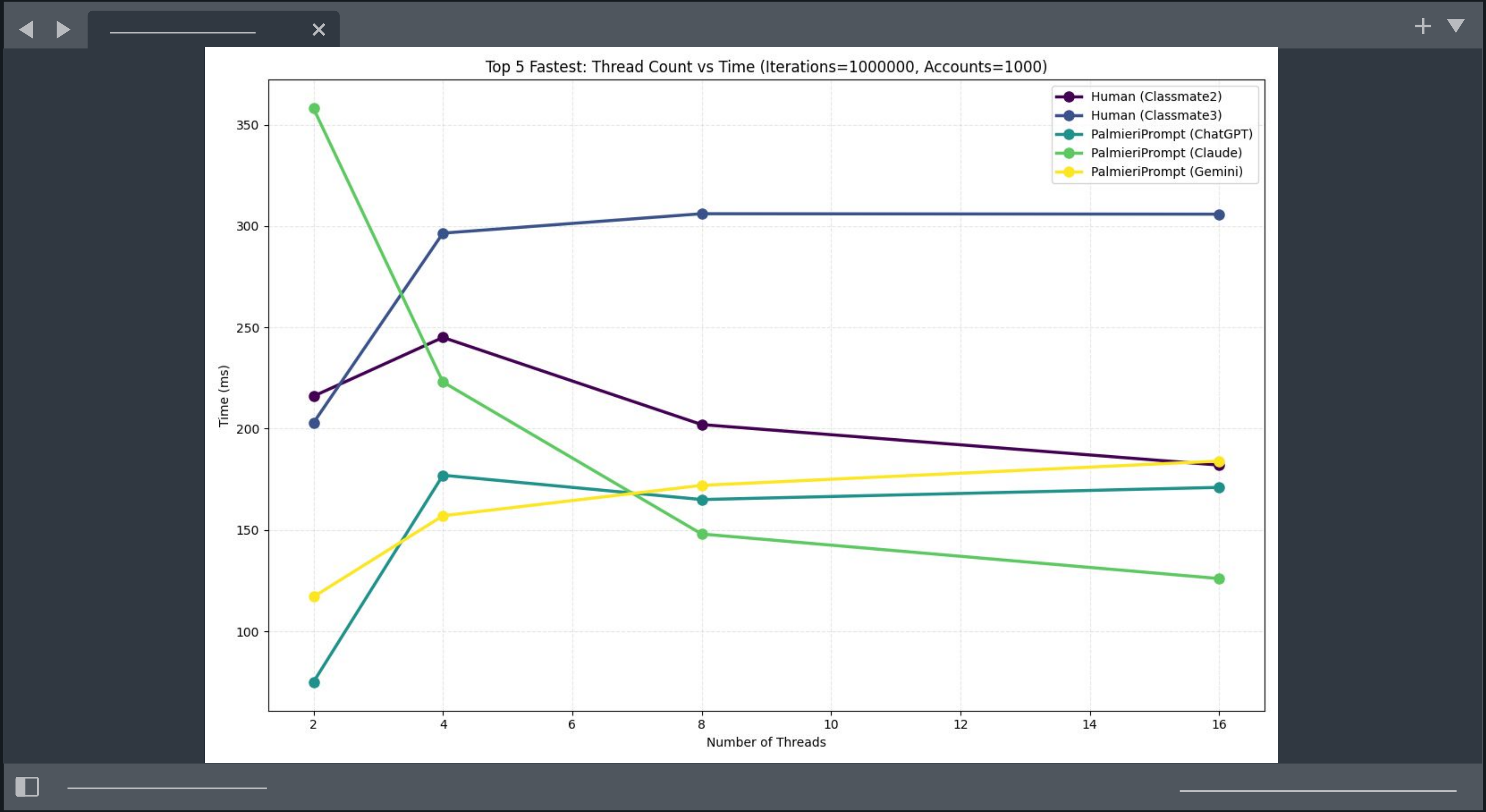
Thread counts: 2, 4, 8, 16
Account counts: 250, 1k, 10k
Iteration counts: 100k, 1mil, 10mil



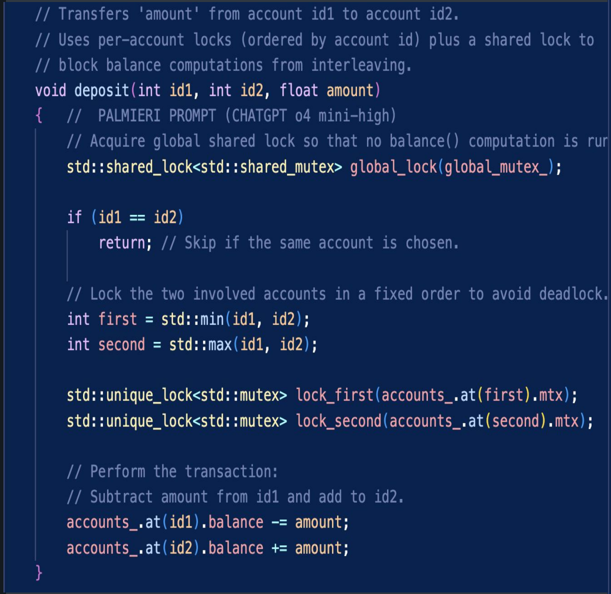
Accounts vs Time (Iterations=1000000, Threads=4)
Time (ms)
10^2
10^3
10^4
Number of Accounts
10^3
10^4
EmirPrompt (ChatGPT)
EmirPrompt (Gemini)
Human (Classmate1)
Human (Classmate2)
Human (Classmate3)
Human (Classmate4)
PalmieriPrompt (ChatGPT)
PalmieriPrompt (Claude)
PalmieriPrompt (Gemini)
RAG (EmirPrompt) (RAG)
RAG (PalmieriPrompt) (RAG)
RAG (StarterPrompt) (RAG)
RAG (StudentPrompt) (RAG)
StarterPrompt (ChatGPT)
StarterPrompt (Claude)
StarterPrompt (Gemini)
StudentPrompt (ChatGPT)
StudentPrompt (Gemini)


Top 5 Fastest: Accounts vs Time (Iterations=1000000, Threads=4)
Human (Classmate2)
Human (Classmate3)
PalmieriPrompt (ChatGPT)
PalmieriPrompt (Claude)
PalmieriPrompt (Gemini)
10^3
Time (ms)
10^3
10^4
Number of Accounts

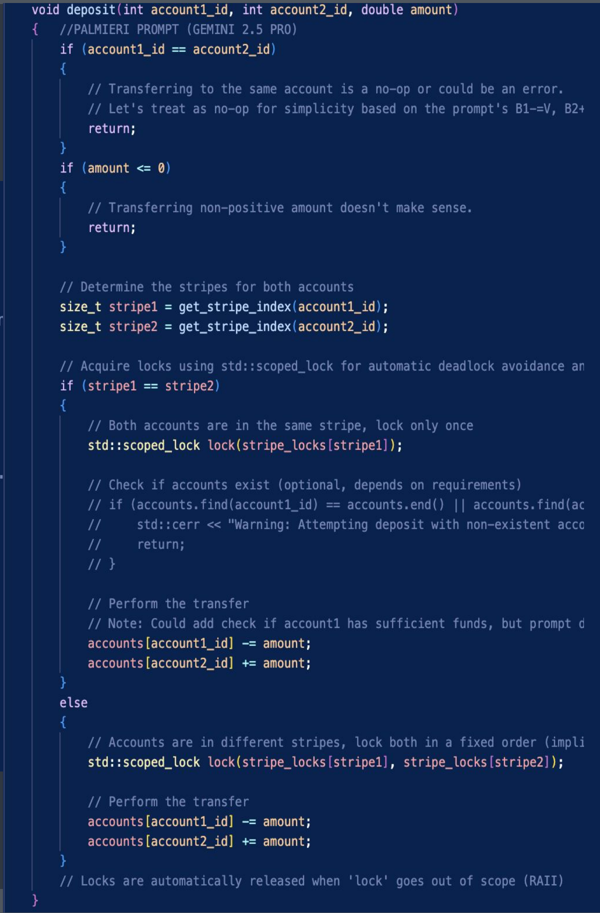

Thread Count vs Time (Iterations=1000000, Accounts=1000)
Time (ms)
0
2000
4000
6000
8000
10000
12000
14000
Number of Threads
2
4
6
8
10
12
14
16
EmirPrompt (ChatGPT)
EmirPrompt (Gemini)
Human (Classmate1)
Human (Classmate2)
Human (Classmate3)
Human (Classmate4)
PalmieriPrompt (ChatGPT)
PalmieriPrompt (Claude)
PalmieriPrompt (Gemini)
RAG (EmirPrompt) (RAG)
RAG (PalmieriPrompt) (RAG)
RAG (StarterPrompt) (RAG)
RAG (StudentPrompt) (RAG)
StarterPrompt (ChatGPT)
StarterPrompt (Claude)
StarterPrompt (Gemini)
StudentPrompt (ChatGPT)
StudentPrompt (Gemini)


Top 5 Fastest: Thread Count vs Time (Iterations=1000000, Accounts=1000)
Human (Classmate2)
Human (Classmate3)
PalmieriPrompt (ChatGPT)
PalmieriPrompt (Claude)
PalmieriPrompt (Gemini)
Time (ms)
350
300
250
200
150
100
Number of Threads
2
4
6
8
10
12
14
16

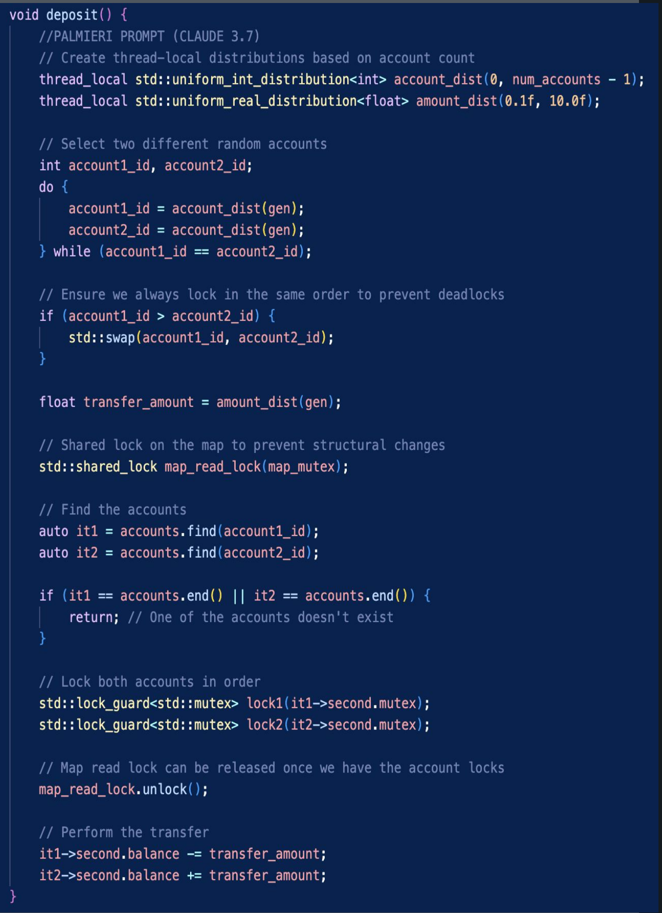
Sample Code



```cpp
// Transfers 'amount' from account id1 to account id2.
// Uses per-account locks (ordered by account id) plus a shared lock to
// block balance computations from interleaving.
void deposit(int id1, int id2, float amount)
{   //  PALMIERI PROMPT (CHATGPT o4 mini-high)
    // Acquire global shared lock so that no balance() computation is rur
    std::shared_lock<std::shared_mutex> global_lock(global_mutex_);

    if (id1 == id2)
        return; // Skip if the same account is chosen.

    // Lock the two involved accounts in a fixed order to avoid deadlock.
    int first = std::min(id1, id2);
    int second = std::max(id1, id2);

    std::unique_lock<std::mutex> lock_first(accounts_.at(first).mtx);
    std::unique_lock<std::mutex> lock_second(accounts_.at(second).mtx);

    // Perform the transaction:
    // Subtract amount from id1 and add to id2.
    accounts_.at(id1).balance -= amount;
    accounts_.at(id2).balance += amount;
}
```



```cpp
void deposit(int account1_id, int account2_id, double amount)
{   //PALMIERI PROMPT (GEMINI 2.5 PRO)
    if (account1_id == account2_id)
    {
        // Transferring to the same account is a no-op or could be an error.
        // Let's treat as no-op for simplicity based on the prompt's B1-=V, B2+
        return;
    }
    if (amount <= 0)
    {
        // Transferring non-positive amount doesn't make sense.
        return;
    }

    // Determine the stripes for both accounts
    size_t stripe1 = get_stripe_index(account1_id);
    size_t stripe2 = get_stripe_index(account2_id);

    // Acquire locks using std::scoped_lock for automatic deadlock avoidance an
    if (stripe1 == stripe2)
    {
        // Both accounts are in the same stripe, lock only once
        std::scoped_lock lock(stripe_locks[stripe1]);

        // Check if accounts exist (optional, depends on requirements)
        // if (accounts.find(account1_id) == accounts.end() || accounts.find(ac
        //     std::cerr << "Warning: Attempting deposit with non-existent acco
        //     return;
        // }

        // Perform the transfer
        // Note: Could add check if account1 has sufficient funds, but prompt d
        accounts[account1_id] -= amount;
        accounts[account2_id] += amount;
    }
    else
    {
        // Accounts are in different stripes, lock both in a fixed order (impli
        std::scoped_lock lock(stripe_locks[stripe1], stripe_locks[stripe2]);

        // Perform the transfer
        accounts[account1_id] -= amount;
        accounts[account2_id] += amount;
    }
    // Locks are automatically released when 'lock' goes out of scope (RAII)
}
```



```cpp
void deposit() {
    //PALMIERI PROMPT (CLAUDE 3.7)
    // Create thread-local distributions based on account count
    thread_local std::uniform_int_distribution<int> account_dist(0, num_accounts - 1);
    thread_local std::uniform_real_distribution<float> amount_dist(0.1f, 10.0f);

    // Select two different random accounts
    int account1_id, account2_id;
    do {
        account1_id = account_dist(gen);
        account2_id = account_dist(gen);
    } while (account1_id == account2_id);

    // Ensure we always lock in the same order to prevent deadlocks
    if (account1_id > account2_id) {
        std::swap(account1_id, account2_id);
    }

    float transfer_amount = amount_dist(gen);

    // Shared lock on the map to prevent structural changes
    std::shared_lock map_read_lock(map_mutex);

    // Find the accounts
    auto it1 = accounts.find(account1_id);
    auto it2 = accounts.find(account2_id);

    if (it1 == accounts.end() || it2 == accounts.end()) {
        return; // One of the accounts doesn't exist
    }

    // Lock both accounts in order
    std::lock_guard<std::mutex> lock1(it1->second.mutex);
    std::lock_guard<std::mutex> lock2(it2->second.mutex);

    // Map read lock can be released once we have the account locks
    map_read_lock.unlock();

    // Perform the transfer
    it1->second.balance -= transfer_amount;
    it2->second.balance += transfer_amount;
}
```

K-Means

Quick Benchmarking

Task: Optimize K-means (sequential baseline vs. multi-threaded)

- Prompts to first “good” response (ignoring clarifying questions)
- “Fix this” iterations to pass compilation/tests
- Max execution time (not average)

Parameters:

Thread count: 16

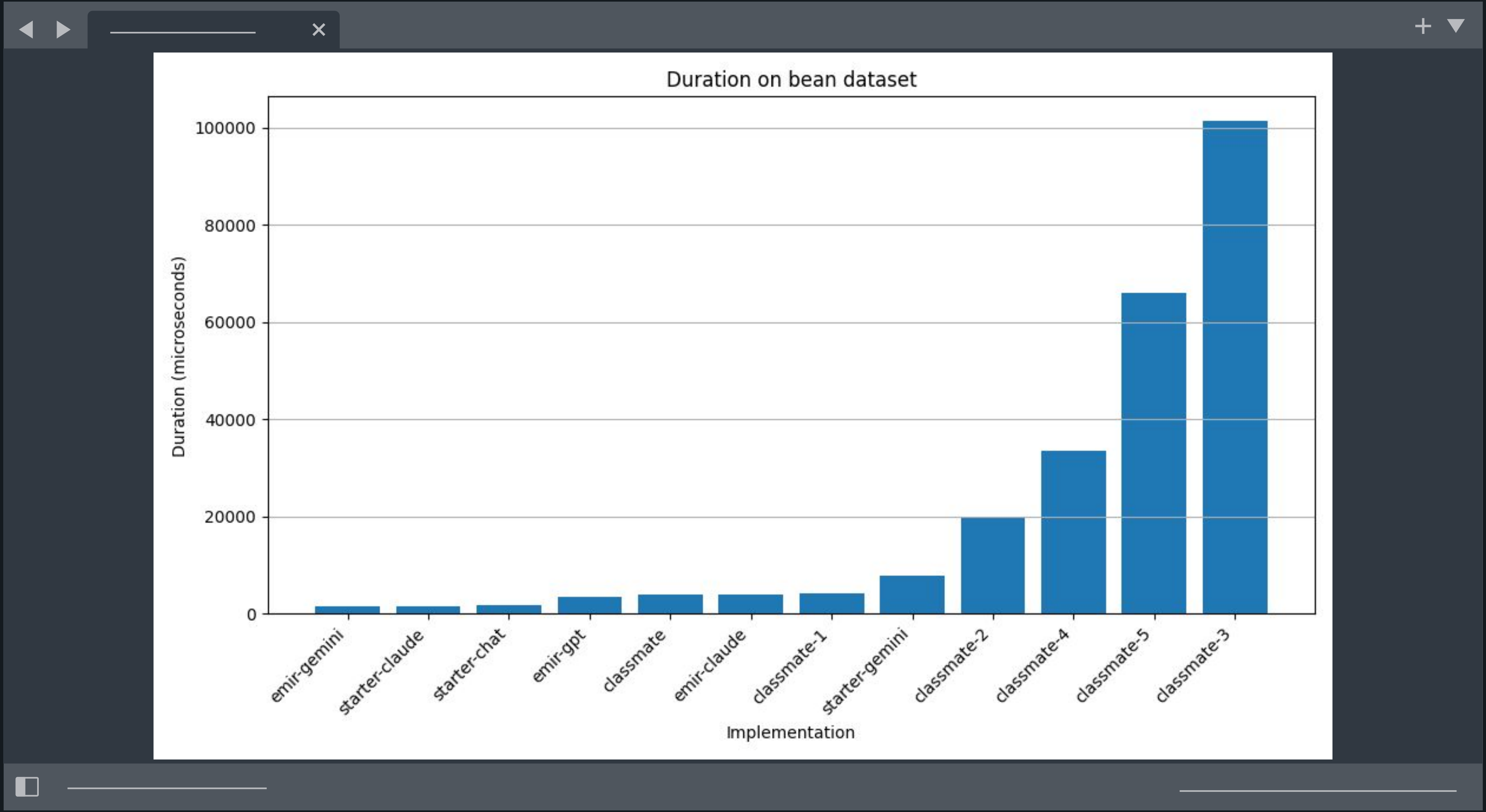
Standardize Cluster Randomizer: **std::mt19937 gen(714);**

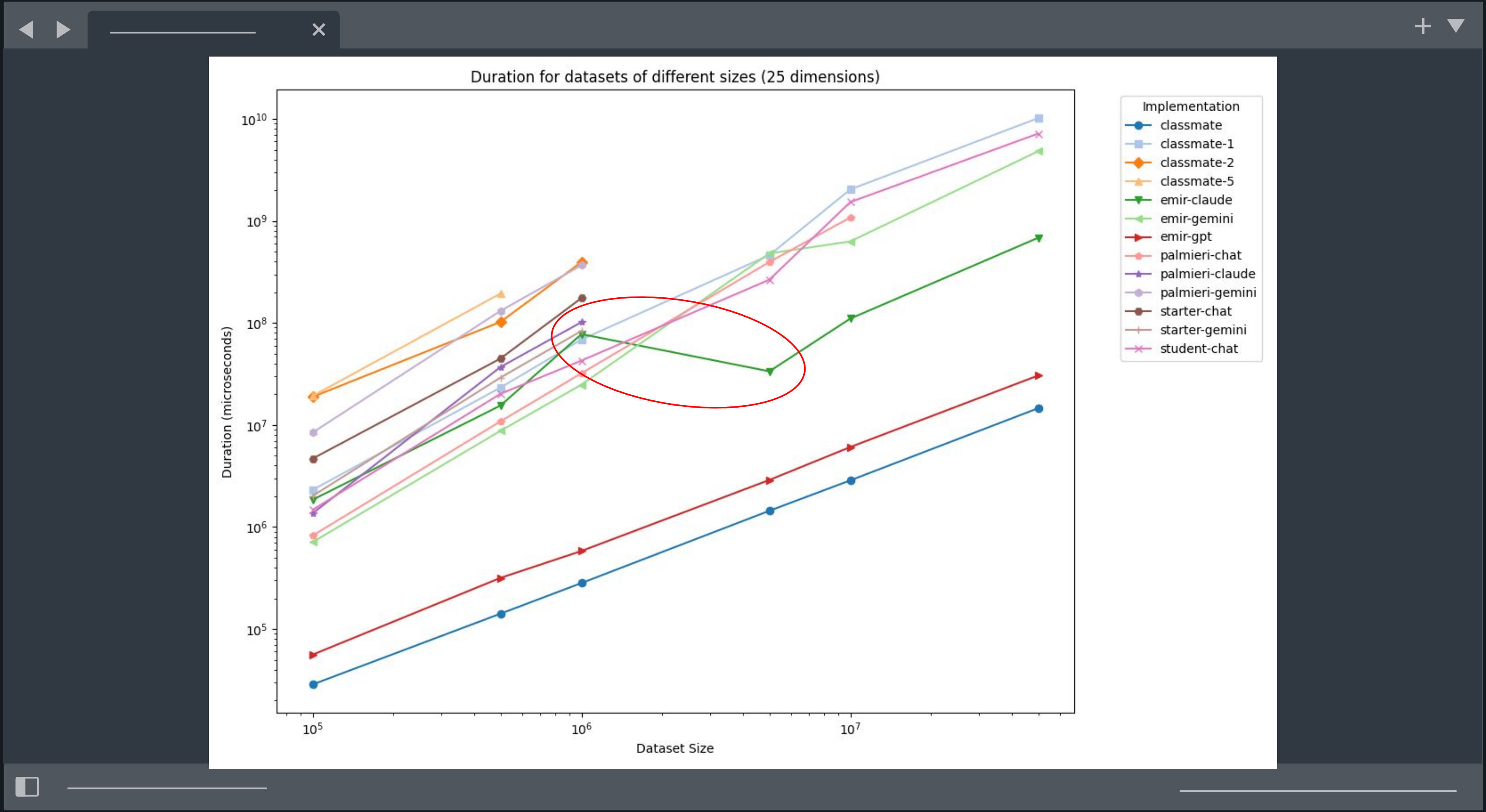
Julian Dataset Breakdown:

Total_points:	
100k, 500k, 1m, 5m, 10m, 50m	
Total_dimensions:	25
K clusters:	20
Max_iterations:	100,000

Drybean Dataset Breakdown:

Total_points:	13,611
Total_dimensions:	16
K clusters:	4
Max_iterations:	50


Duration on bean dataset
Duration (microseconds)
100000
80000
60000
40000
20000
0
emir-gemini
starter-claude
starter-chat
emir-gpt
classmate
emir-claude
classmate-1
starter-gemini
classmate-2
classmate-4
classmate-5
classmate-3
Implementation


Duration for datasets of different sizes (25 dimensions)
Duration (microseconds)
10^5
10^6
10^7
10^8
10^9
10^10
Dataset Size
10^5
10^6
10^7
Implementation
classmate
classmate-1
classmate-2
classmate-5
emir-claude
emir-gemini
emir-gpt
palmieri-chat
palmieri-claude
palmieri-gemini
starter-chat
starter-gemini
student-chat

“That’s not as fast as it can be”

Problem:

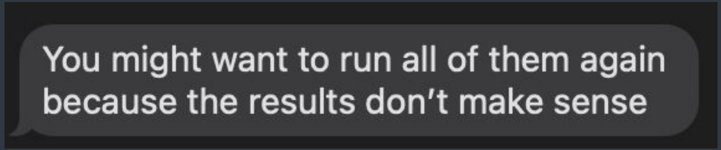
- Benchmarking showed slower execution times, especially on very large datasets (1M+ points).

Insight from Claude:

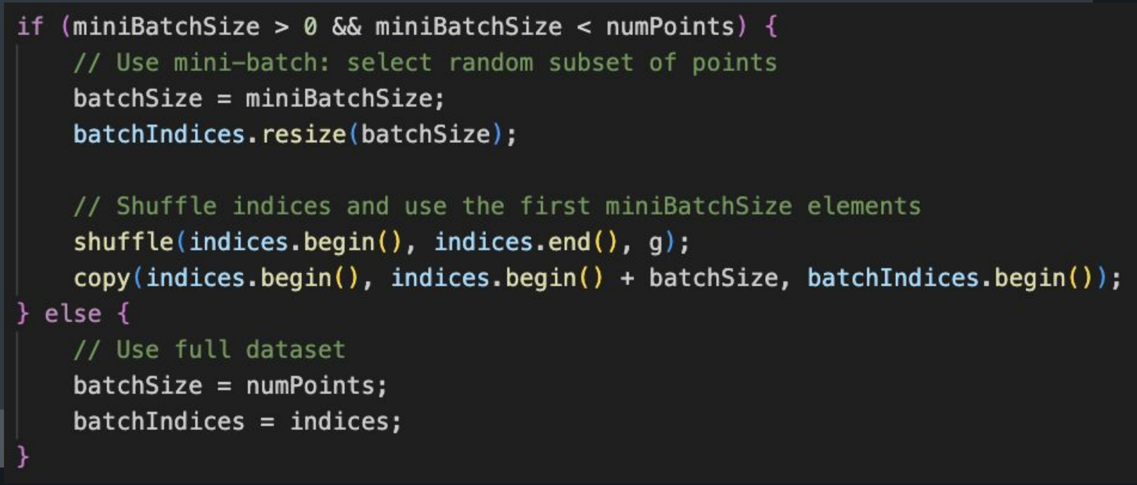
- Introduced mini-batching, a method of processing data in smaller subsets rather than the entire dataset simultaneously.

Tradeoff:

- Mini-batching will introduce slight variations in accuracy and convergence.

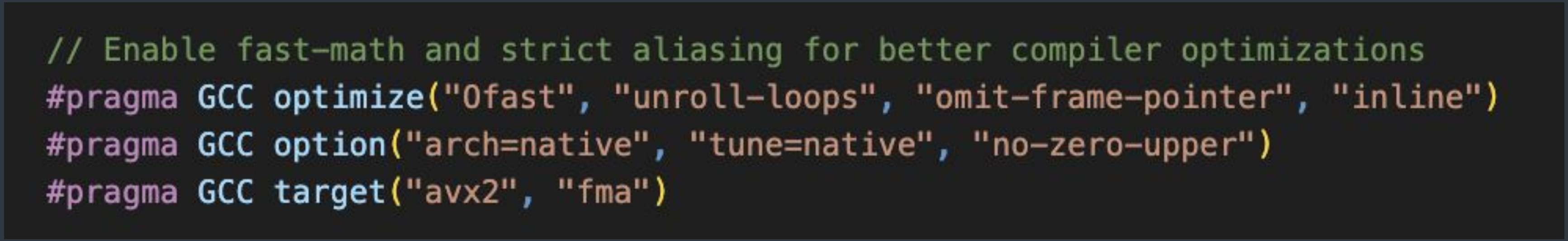


You might want to run all of them again because the results don’t make sense

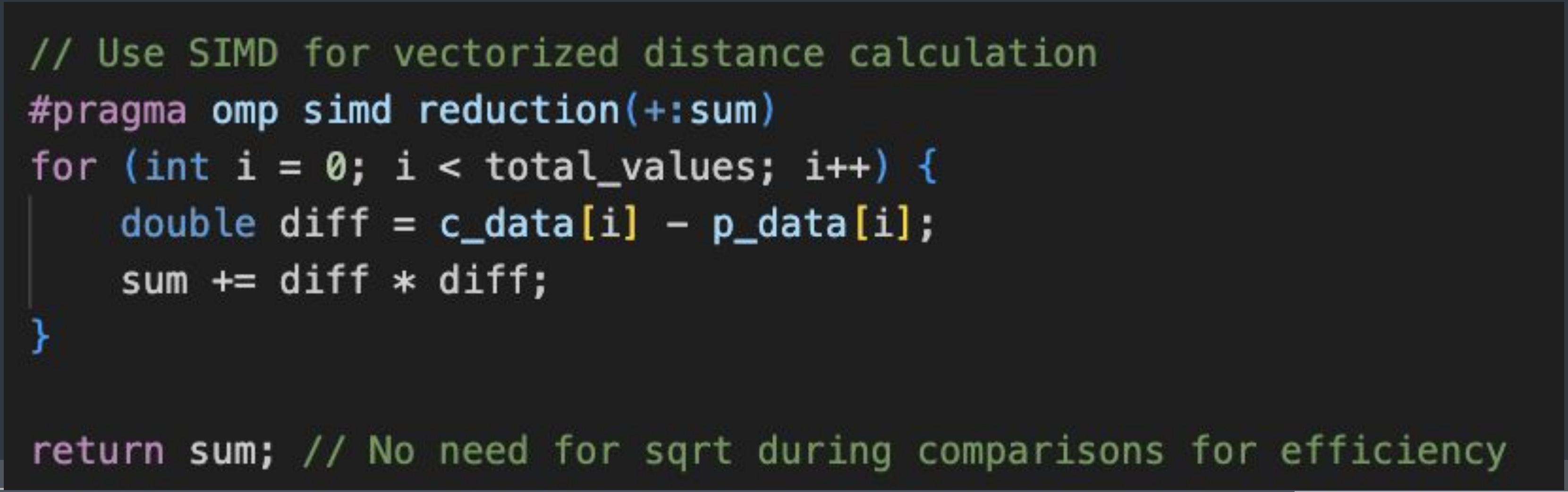


```
if (miniBatchSize > 0 && miniBatchSize < numPoints) {
    // Use mini-batch: select random subset of points
    batchSize = miniBatchSize;
    batchIndices.resize(batchSize);

    // Shuffle indices and use the first miniBatchSize elements
    shuffle(indices.begin(), indices.end(), g);
    copy(indices.begin(), indices.begin() + batchSize, batchIndices.begin());
} else {
    // Use full dataset
    batchSize = numPoints;
    batchIndices = indices;
}
```

Emir-GPT Optimizations:

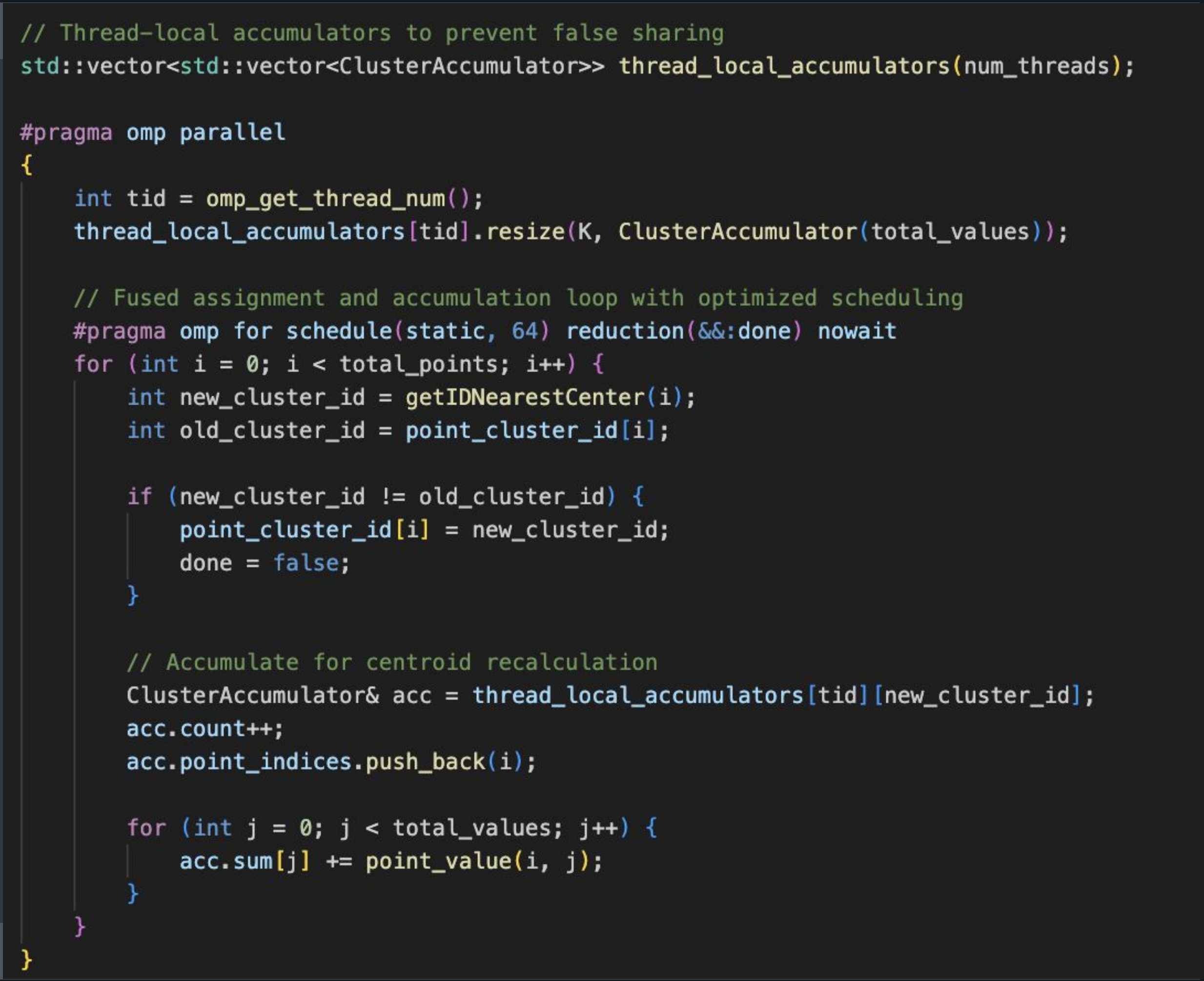


```
// Enable fast-math and strict aliasing for better compiler optimizations
#pragma GCC optimize("Ofast", "unroll-loops", "omit-frame-pointer", "inline")
#pragma GCC option("arch=native", "tune=native", "no-zero-upper")
#pragma GCC target("avx2", "fma")
```



```
// Use SIMD for vectorized distance calculation
#pragma omp simd reduction(+:sum)
for (int i = 0; i < total_values; i++) {
    double diff = c_data[i] - p_data[i];
    sum += diff * diff;
}

return sum; // No need for sqrt during comparisons for efficiency
```

Emir-GPT Optimizations:



```cpp
// Thread-local accumulators to prevent false sharing
std::vector<std::vector<ClusterAccumulator>> thread_local_accumulators(num_threads);

#pragma omp parallel
{
    int tid = omp_get_thread_num();
    thread_local_accumulators[tid].resize(K, ClusterAccumulator(total_values));

    // Fused assignment and accumulation loop with optimized scheduling
    #pragma omp for schedule(static, 64) reduction(&&:done) nowait
    for (int i = 0; i < total_points; i++) {
        int new_cluster_id = getIDNearestCenter(i);
        int old_cluster_id = point_cluster_id[i];

        if (new_cluster_id != old_cluster_id) {
            point_cluster_id[i] = new_cluster_id;
            done = false;
        }

        // Accumulate for centroid recalculation
        ClusterAccumulator& acc = thread_local_accumulators[tid][new_cluster_id];
        acc.count++;
        acc.point_indices.push_back(i);

        for (int j = 0; j < total_values; j++) {
            acc.sum[j] += point_value(i, j);
        }
    }
}
```

Cuckoo

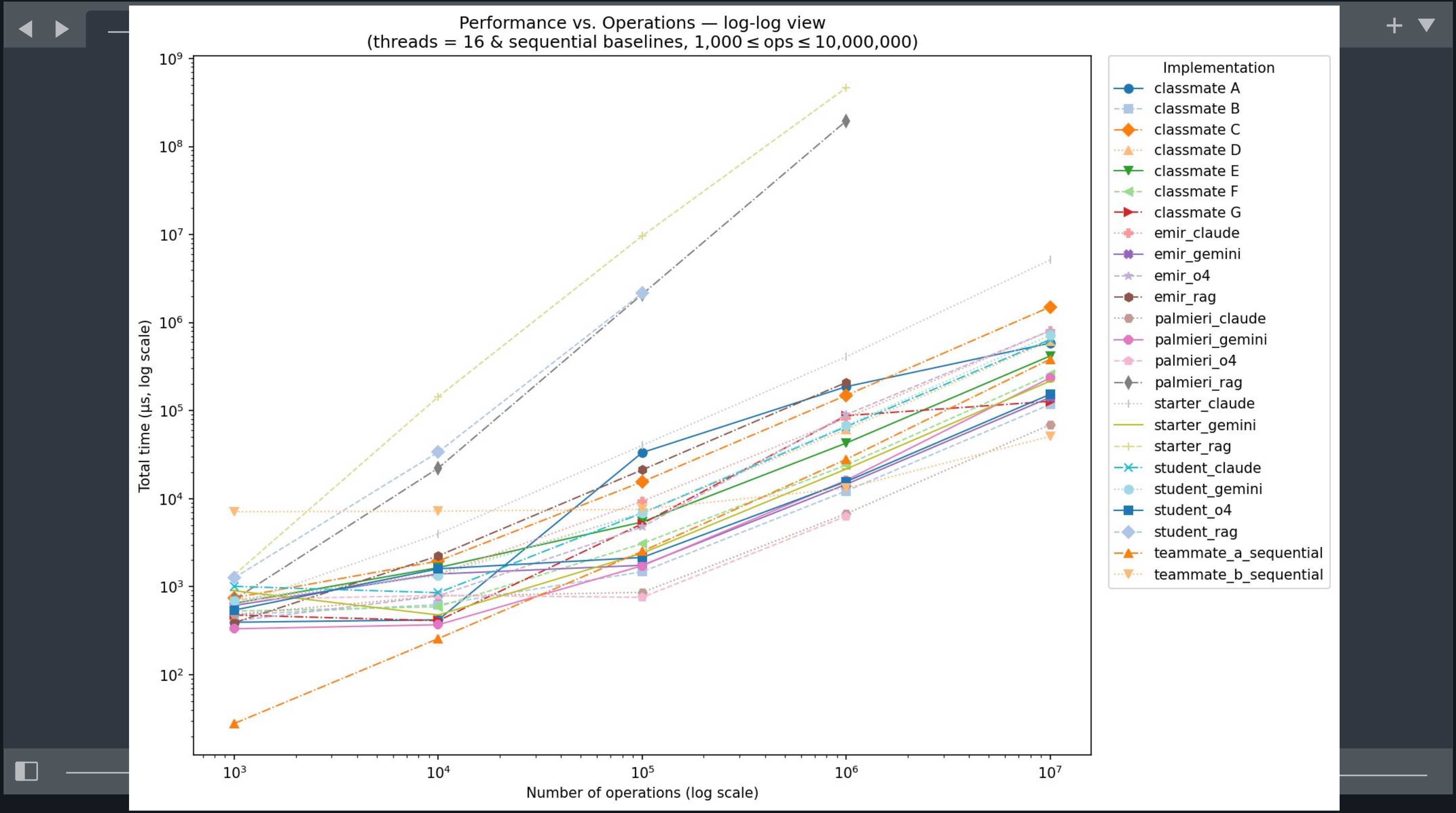
Quick Benchmarking

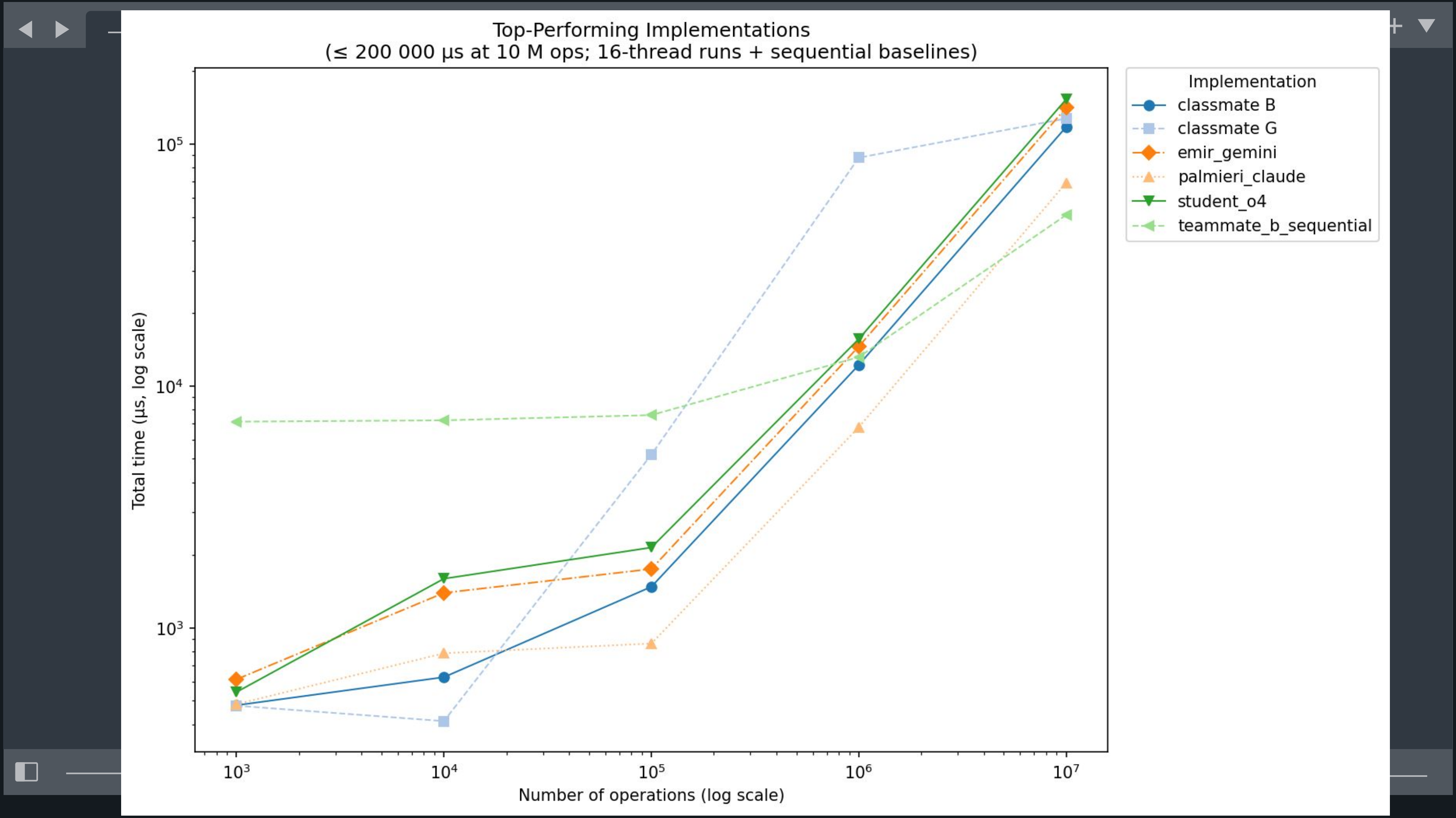
Task: Implement Cuckoo Hashset (sequential baseline vs. multi-threaded)

- Prompts to first “good” response (ignoring clarifying questions)
- “Fix this” iterations to pass compilation/tests
- Total execution time

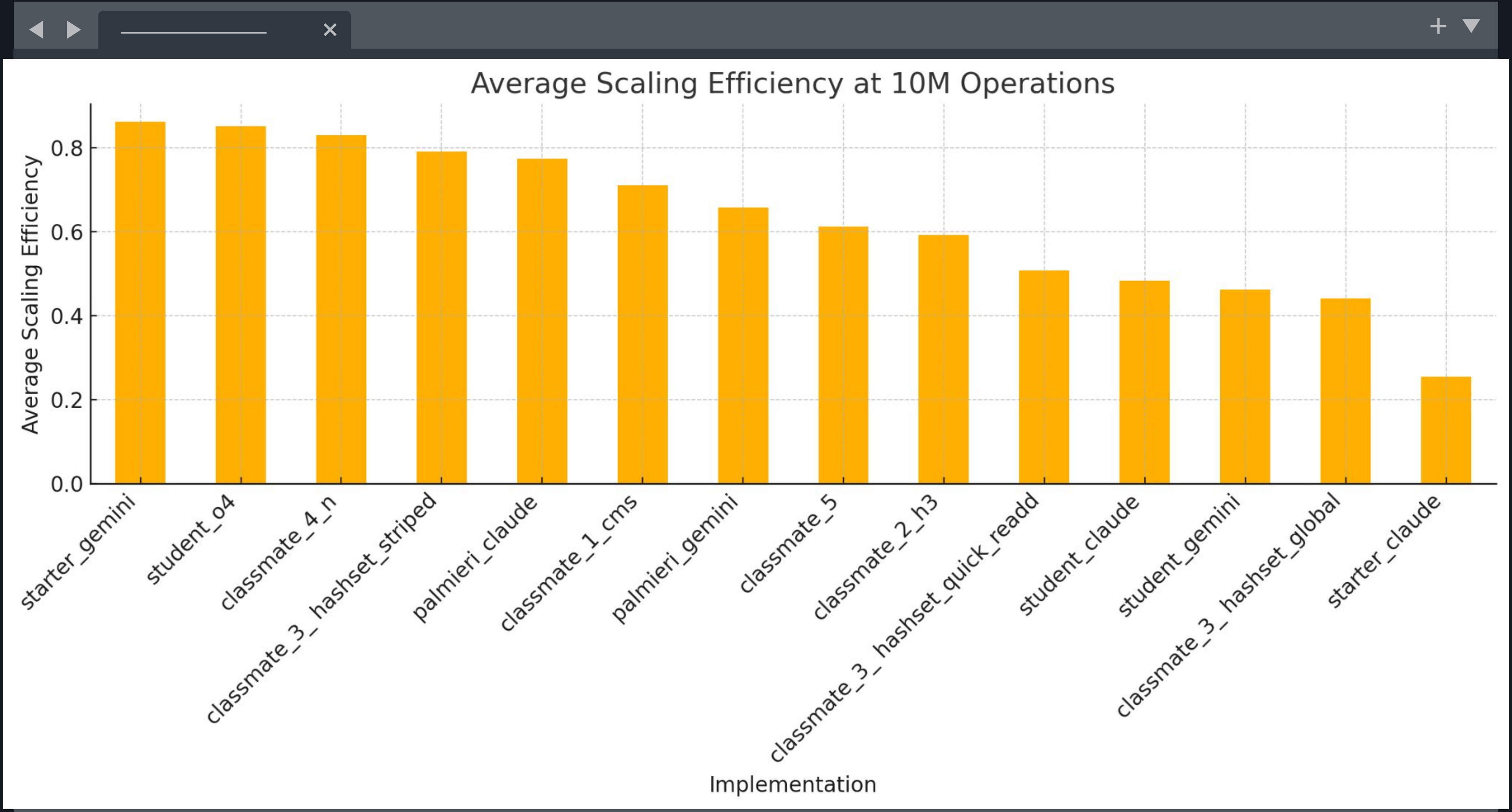
Parameters for Cuckoo:

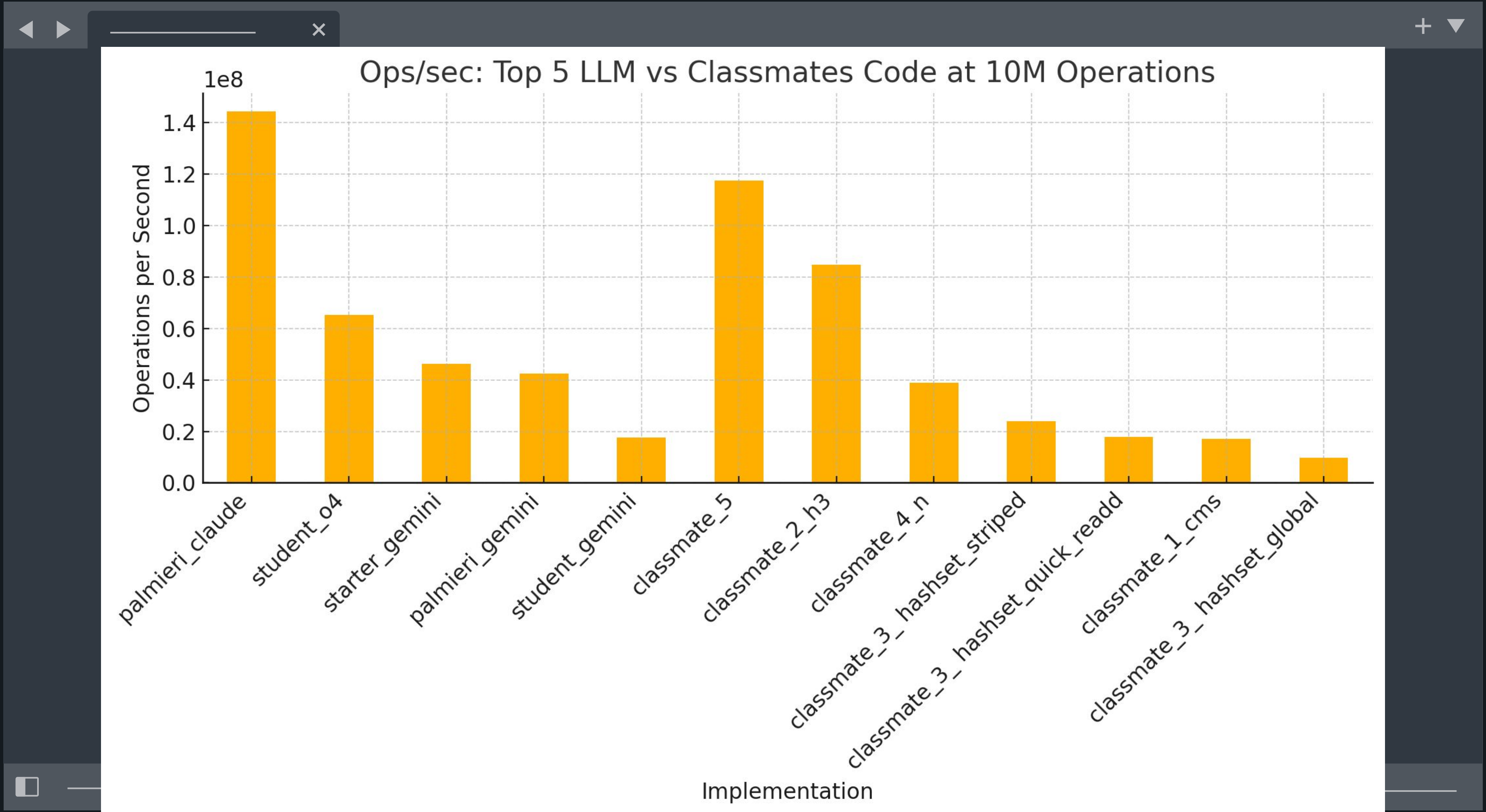
Initial Capacity:	1,000,000 integers
Initial Size:	500,000 integers
Thread counts:	2, 4, 8, 16, 24 threads
Operation counts:	1k, 10k, 100k, 1mil, 10mil


Performance vs. Operations — log-log view
(threads = 16 & sequential baselines, 1,000 ≤ ops ≤ 10,000,000)
Total time (µs, log scale)
10^9
10^8
10^7
10^6
10^5
10^4
10^3
10^2
10^3
10^4
10^5
10^6
10^7
Number of operations (log scale)
Implementation
classmate A
classmate B
classmate C
classmate D
classmate E
classmate F
classmate G
emir_claude
emir_gemini
emir_o4
emir_rag
palmieri_claude
palmieri_gemini
palmieri_o4
palmieri_rag
starter_claude
starter_gemini
starter_rag
student_claude
student_gemini
student_o4
student_rag
teammate_a_sequential
teammate_b_sequential



Top-Performing Implementations
(≤ 200 000 µs at 10 M ops; 16-thread runs + sequential baselines)


Average Scaling Efficiency at 10M Operations
Average Scaling Efficiency
0.0
0.2
0.4
0.6
0.8
starter_gemini
student_o4
classmate_4_n
classmate_3_ hashset_striped
palmieri_claude
classmate_1_cms
palmieri_gemini
classmate_5
classmate_2_h3
classmate_3_ hashset_quick_readd
student_claude
student_gemini
classmate_3_ hashset_global
starter_claude
Implementation

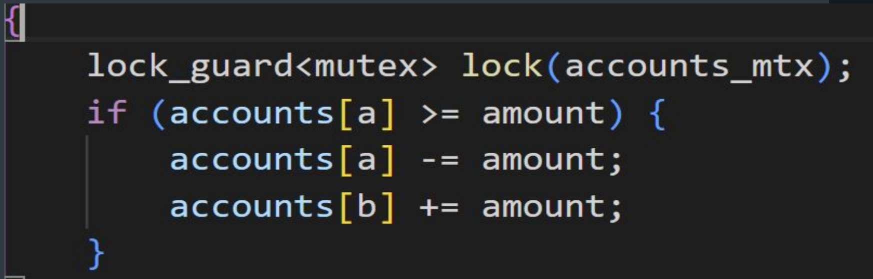


Ops/sec: Top 5 LLM vs Classmates Code at 10M Operations
1e8
Operations per Second
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
palmieri_claude
student_o4
starter_gemini
palmieri_gemini
student_gemini
classmate_5
classmate_2_h3
classmate_4_n
classmate_3_ hashset_striped
classmate_3_ hashset_quick_readd
classmate_1_cms
classmate_3_ hashset_global
Implementation

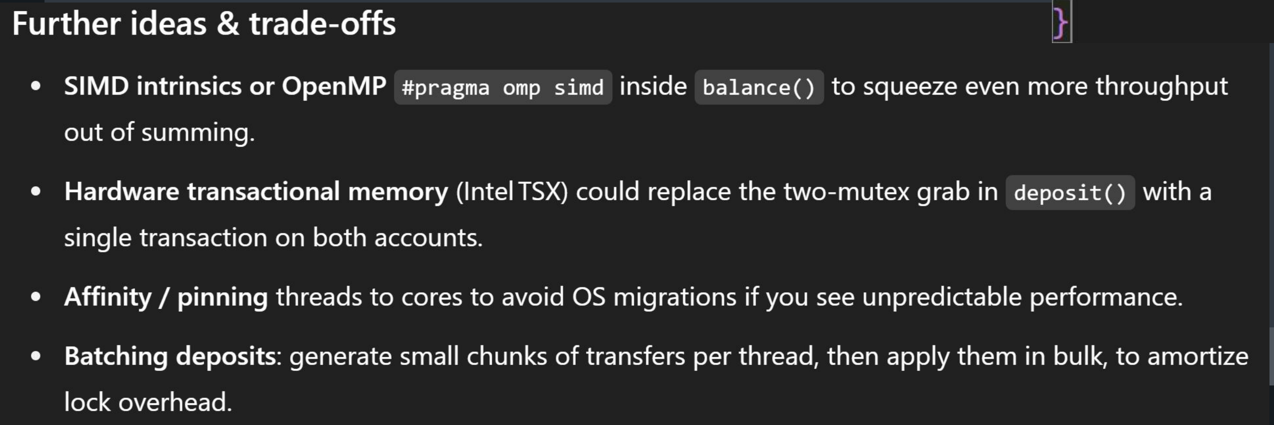
RAG

Interesting Results (Bank)

- Pre-computing rands()
- Compiler hints for vectorization (#pragma GCC ivdep)
- HTM (_xbegin/end)
- Thread local storage
- Padding mutex/locks to the cachline



```
{
    lock_guard<mutex> lock(accounts_mtx);
    if (accounts[a] >= amount) {
        accounts[a] -= amount;
        accounts[b] += amount;
    }
}
```



Further ideas & trade-offs

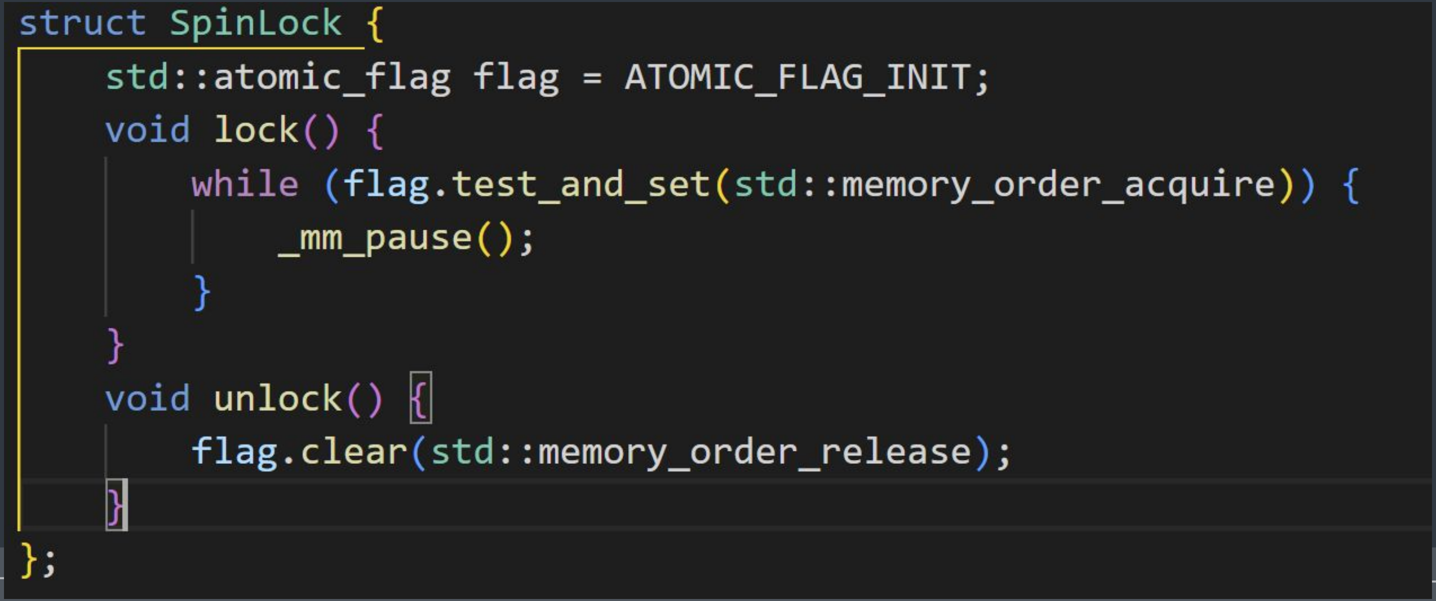
- **SIMD intrinsics or OpenMP** `#pragma omp simd` inside `balance()` to squeeze even more throughput out of summing.
- **Hardware transactional memory** (Intel TSX) could replace the two-mutex grab in `deposit()` with a single transaction on both accounts.
- **Affinity / pinning** threads to cores to avoid OS migrations if you see unpredictable performance.
- **Batching deposits**: generate small chunks of transfers per thread, then apply them in bulk, to amortize lock overhead.

Interesting Results (K-Means)

/ Nothing!

Interesting Results (Cuckoo)

- Utilizing prime numbers for hash functions
- Inheritance
- Striping



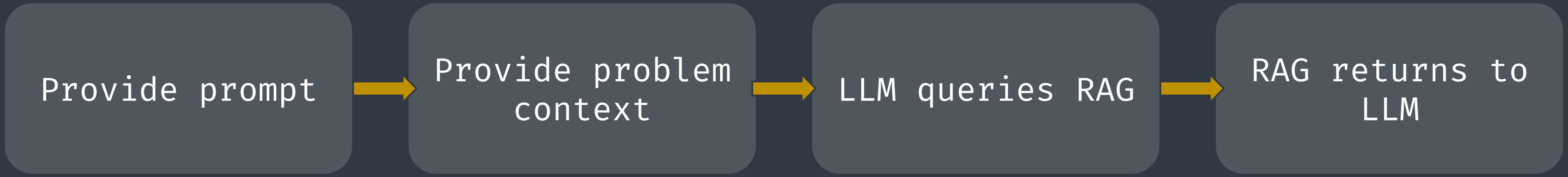
```
struct SpinLock {
    std::atomic_flag flag = ATOMIC_FLAG_INIT;
    void lock() {
        while (flag.test_and_set(std::memory_order_acquire)) {
            _mm_pause();
        }
    }
    void unlock() {
        flag.clear(std::memory_order_release);
    }
};
```

Problems with the RAG

/ Simply providing entire prompt

/ Character lengths

Designing a New Procedure



Provide prompt

Provide problem context

LLM queries RAG

RAG returns to LLM

LLM Problems

- Not implementing, just suggesting how to do it
- Outputting code with errors
  - Copying and moving mutexes
  - Forgetting import statements
  - Bank - rounding/precision errors (float vs double)
- Forgetting to follow the actual HW instructions
- Cuckoo - randomly inserting when failed to insert
- Cuckoo - recursive resize, infinite stall & failure

What’s Next?

- Optimize RAG
- Optimize Prompting
  - How extensive do prompts need to be?
  - Build out an all an encompassing prompt
- Keep track of # of prompts for “good output” and “fix this”

Thank you!

Questions?