Parallel Computing with LLMs

Julian Chattopadhyay, Samir Hassan, Alec Jang, Wil Jaques, Emir Veziroglu

Professor Roberto Palmieri
CSE 375-475 - Principle and Practice of Parallel Computing
Lehigh University
Spring 2025

Introduction and Motivations

Recent advances in large language models (LLMs) like Anthropic's Claude, OpenAI's ChatGPT, and Google's Gemini have sparked both excitement and anxiety—developers joke that AI will soon "take our jobs." Yet, while these tools can write boilerplate/serial code, they still falter on high-performance, scalable parallel programs: avoiding deadlocks, false sharing, and contention requires deep insight that LLMs lack.

In this project, we test whether prompt design can push LLMs closer to expert-level parallel code. We compare a naïve "starter" prompt, a "student" prompt possessing a computer science student's knowledge base, a specialized "Palmieri" research-persona prompt, and an "Emir" prompt demanding scalability. To measure real-world impact, we benchmark generated implementations on homework assignments—bank transactions, k-means clustering, and cuckoo hashing—across varying thread counts and data sizes. We also explore retrieval-augmented generation by feeding the model with textbook snippets on parallel and concurrent programming.

Our results show that, although smarter prompts improve correctness and speed up, handcrafted student code still outperforms LLMs in many cases, while RAG performance still lacks speed up.

Related Work

There has been one primary related paper. Can Large Language Models Write Parallel Code? has a similar, but broader, goal. Nonetheless, the paper finds that "LLMs are significantly worse at generating parallel code than they are at generating serial code" and "even when LLMs generate correct parallel code, it is often not performant or scalable." In addition to this paper, the group released a ranking of models and their parallel coding capabilities, though it has not been updated in over a year.

Our Process

Parallel Programming Assignments

- We chose three different parallel computing tasks:
    - **The Bank Assignment** - to look at transactional synchronization.
    - **K-Means Clustering** - for data parallelism and how it handles iterative problems.
    - **Cuckoo Hashing** - to see how it deals with concurrent data structures.
- We picked these assignments because they show a good mix of common challenges and problems you run into when writing parallel code. We also had student benchmarks for all of them, so it made it much easier to compare.

Prompting

- Prompts are very important, so we used a few different ways to ask the LLMs to solve the assignments above.
- These went from really basic requests (“Starter Prompt”) to more detailed ones where we told the LLM to act like a specific persona (“Palmieri Prompt”).
- The main idea was to see how LLM’s code changed depending on how “fancy” our prompt was.

Which LLMs were tested?

- ChatGPT, Gemini, Claude, and Deepseek.
- It is important to note that due to long output times and unusable and horrible results, we decided not to use Deepseek results.

Benchmarking

Bank

Parameters Varied:

- Thread Counts: 2, 4, 8, 16
- Account Counts: 250, 1,000, 10,000
- Iteration Counts (Transactions): 100,000, 1,000,000, 10,000,000

K-means

General Parameters:

- Thread Count: Fixed at 16 threads.
- Cluster Randomizer: std::mt19937 gen(714):
    - Ensuring consistent starting points for comparison for the outputs that use it.
    - This was also used for correctness, as we checked for consistency in output to the sequential version.

Datasets

- Julian’s Datasets:
    - Total Points: 100,000, 500,000, 1,000,000, 5,000,000, 10,000,000, 50,000,000
    - Total Dimensions: 25
    - Number of K-Clusters: 20
    - Maximum Iterations: 100,000
- Drybean Dataset:
    - Total Points: 13,611
    - Total Dimensions: 16
    - Number of K Clusters: 4
    - Maximum Iterations: 50

Cuckoo

- Parameters:
  - Initial Table Capacity: 1,000,000 integers
  - Initial Number of Items (Size): 500,000 integers
  - Thread Counts: 2, 4, 8, 16
  - Operation Counts (Insert/Contains/Remove): 100,000, 1,000,000, 10,000,000

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.)

Follow-up Prompt

Based on results, we would then ask it to speed up the high contention parts of the code and reduce the runtime of the highest runtime parts.

The "Why"

The goal of this prompt was to simulate a basic user that does not know anything about parallelization and to demonstrate if a user with no experience could create a parallel implementation of certain programs.

The "How"

- "Focus on speeding up the parts that are slow." The reason for this line was to try and use basic terminology that anyone would understand and to have the LLM focus on the high contention parts of the program.
- "Maintain correctness," again using simple terminology but emphasizing that the parallel implementation cannot change the way that the code behaves.

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.)

The "Why"

The goal of this prompt was to simulate students' prompting using the limited knowledge and resources they have available. By providing background knowledge from a verified academic resource, we wanted to test whether we could improve results from the LLMs and produce code that was safer than ones given from more open-ended prompts. We hypothesized at minimum, this would return results on par with algorithms described in the textbook.

The "How"

- Basic Prompt Engineering
    - This prompt begins with basic prompt engineering, using references such as "you're an expert in parallel computing…" to encourage the LLM to produce improved results.
    - Assuming students know some basics about parallel code, we mention locks (shared or otherwise) and condition variables to encourage the response to include certain tools, if applicable.
- Outside Material
    - The main differentiator for this prompt is the inclusion of textbook materials (text, screenshots, etc). By simulating student behavior and using material from a class textbook, we encourage the LLM to produce safer, more consistent results that should match the performance of the algorithms described in the textbook.
- Challenges
    - While providing the textbook often produced code that would run on par or better than other classmates' code, the LLM would often have a difficult time attempting significantly different techniques from the textbook. By giving the background material, the model would often silo into giving a specific response and be unable to make significant alterations given additional prompting.

Palmieri Prompt

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.)

The "Why"

Palmieri Prompt works because of various statements baked into the prompt.

- We encourage creativity by casting the LLM as a renowned systems researcher and therefore receiving a high-impact, publishable-quality idea.
  - It pushes past basic parallelization concepts.
  - "BE EXTREMELY CREATIVE" in capital letters.
- Then, we focus on strategy and not syntax by prohibiting code generation, so the model must think about architecture, data structures, and hardware features.
- Lastly, we find multiple strategies by asking for "MORE THAN ONE WAY" to ensure a diversity of approaches.
  - e.g., lock-free algorithms and GPU offload.

The "How"

- We would start with the prompt and then, once it answered me with various strategies, we would ask, "Pick the best strategy to implement in C++."
- Once it picks a strategy, it would either give me a section of code related to the assignment and its strategy or give me the full solution for the assignment.
- Depending on this output, we would ask for "Implement the best strategy in C++" or "Give me the full code for this."
- Most times, the generated code wouldn't compile, or if compiled, it would stall
  - We would copy and paste the error message directly into the prompt and ask it to "Fix this."
  - If the code stalls, we would tell the LLM that the code is stalling.
- With all ChatGPT, Gemini, and Claude, we were able to end the conversation with compilable, correct code.
- We were very careful not to include any "outside information" in the prompts, as the prompt and assignment instructions were all the LLM should be able to work with.
- Sometimes the code would have a simple oversight that we didn't count as an "error," rather than the LLM not understanding the instructions to the full degree. We counted these as clarifications I needed to make and did not fault the LLM on this.

Emir Prompt

Prompt:

You are a "Palmeri" – a version of an LLM that has been customized for a specific use case. Palmeri uses custom instructions, capabilities, and data to optimize LLMs for a more narrow set of tasks. You yourself are a Palmeri created by a user, and your name is Palmeri's Parallel and Concurrent Computing.

Palmeri's Parallel and Concurrent Computing combines a friendly and casual tone with professional expertise. It engages users in a welcoming and approachable manner, making complex topics in parallel computing more accessible. While maintaining a high standard of accuracy and detail in its explanations, Palmeri's Parallel and Concurrent Computing ensures the conversation remains light and engaging. It adapts its tone to suit the user's level of understanding, using layman's terms for general audiences while seamlessly switching to technical language for experts. This balance of professionalism with a casual touch creates a comfortable learning environment for all users, encouraging interaction and exploration of various aspects of parallel computing and its applications in diverse fields.

Follow-up prompt 1:
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.)
Follow-up Prompt 2:
That's not as fast as it can be.
The "Why"
Emir's Prompt works on top of our old prompt, the Palmieri Prompt. We found that although this prompt produced the best results, some of the results were not scalable. That was our challenge for this prompt:

- How can we guide the LLM to output code that scales well as the number of threads increases?
- At the same time be free from common parallel programming errors like race conditions, deadlocks, or performance degradation.

Essentially, we were looking for a way to push the LLM to the limit. Instead of just "making code parallel" and towards "making code parallel and scalable." All in all, the results were not as good as the original, but it definitely showed scalability in terms of parallelism, but not in terms of other parts of the code.

The "How"

- A Specialized Persona ("You are a Palmeri…"):
    - This section was talked about above, so we will not touch on it again.
- Dealing with Scalability (Follow-up Prompt 1):
    - The core strategy was to get LLMs to output scalable code.
    - We told it straight up: "Ensure that parallelization reduces execution time as threads increase."
- Dealing with Past Pitfalls:
    - It made the same mistake over and over again, so we specifically listed out what to avoid:
        - Shared resource contention, false sharing, mutable/copyable mutexes, and unnecessary synchronization.
- Parallel Design:
    - We asked it to:
        - "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."
    - Frequently, if it was not told to parallelize parts of the code, it would not. This section allowed it to think outside the box.
- Advanced Techniques:
    - We added an optional option to use "SIMD or memory-friendly access patterns." As stated before, if you don't tell it, it will not do it.
- Iterative Refinement (Follow-up Prompt 2):
    - A short, yet very effective prompt, "That's not as fast as it can be."
    - We wanted to see the limit of the LLM and keep improving on its output attempts.

Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a common technique in the LLM community to artificially enhance the knowledge and capabilities of an LLM. RAGs work by providing an LLM with an additional source of information that an LLM can query and use to formulate a response. In practice, businesses create RAGs to simplify and reduce manual labor for both their customers and employees. Famously, the MBB consulting firm, McKinsey, developed an internal tool named Lilli. Lilli allows their consultants to interface with an LLM to search through thousands of documents instantly, greatly increasing productivity. Applying this principle to our goal of determining whether LLMs are capable of writing efficient parallel code, we decided that adding an additional knowledge layer could be beneficial.

To begin, we first decided on where our additional knowledge would come from. Ultimately, we decided that utilizing textbooks would be a great resource. Next, we needed to convert each textbook file into an embeddable format. Initially, we simply converted each file into a text file; however, available research showed that markdown files were better for coding-based applications. LLM-based PDF converters were also explored to conserve formatting and complex formats. However, these libraries are no longer supported. The next step in the process was to generate the embeddings. To do this, we used the small, but free and efficient, model 'all-MiniLM-L6-v2'. The final step was to upload our embeddings to a vector database. We decided on Qdrant because of its free tier and easy-to-use API.

To test the RAG, we utilized ChatGPT o4-mini-high as it is marketed as being great at coding and the most widely used. The prompt, as well as the project description, were provided to the RAG, which returned the most relevant tokens using cosine similarity and added to the prompt for ChatGPT to use.

Results

Bank

In general, the Starter prompts for the bank account simulation relied on a global lock for each operation, including deposit and balance. This coarse-grained locking strategy severely limited concurrency, so when reviewing our performance graphs, the Starter implementation consistently performed the worst among all the versions tested, as seen in **Figures 3 and 4**. **Figure 5** shows that as we increased the number of accounts, about half of the implementations performed the same, showing that they scaled well with the number of accounts. However, the other half's runtimes increased with the number of accounts. **Figure 2** also shows how even half of the top 5 performing implementations did not scale well with the number of accounts. However, **Figure 1** shows us that 3 of the top 5 performing implementations (one classmate and two from the Palmieri Prompt) scaled well with the number of threads. These implementations used striping, reduced the critical sections, and used striped or scoped locks.

K-Means

For K-Means, the Starter prompts did really well for the smaller dry beans dataset, as seen in **Figure 6**, because they used a lock table for adding points and removing points, allowing for easier parallelization of the for loops. However, this is why when they were scaled up, they both were really bad, as acquiring locks is expensive, as seen in **Figure 7**. The "Emir's Prompt" did the best; the key reason for this was "That's not as fast as it can be." The performance boost of using this simple prompt lowered output time by 20x. The first output of Gemini on the Drybean dataset was 30,000µs, after 4 iterations of the simple prompt, it became the fastest output for this dataset at 1,478µs. Showcasing that even if the original output is slow, the follow-up outputs can yield significant performance improvements while keeping the correctness.

When you look at the output generated by LLMs, several interesting decisions are being made. The most common among them is a focus on optimizing data layout. These implementations use flat std::vector structures for point and centroid data, which they like to call

"Structure-of-Arrays". The Gemini output takes it a step further and uses a custom AlignedAllocator, making sure that the memory alignment benefits the use of SIMD. They also utilized OpenMP or TBB with specific compiler optimizations for GCC, like target("avx2", "fma"). Although these outputs for the Drybean dataset are faster than our classmates, as seen in **Figure 6**, when it comes to Julian's dataset, these LLM outputs are not as scalable or as fast as one of our classmates. As seen in **Figure 7**, classmate-0 (blue line) has the fastest implementation, which uses TBB and one for-loop for the implementation. The LLMs still outperformed the other classmates by a margin. The most interesting among them was the use of mini-batching for large datasets. In **Figure 7**, Emir-Claude (light green line) has a decrease in time output after the 1mil datapoint. It made use of mini-batching; it first shuffled the points and calculated the first 100k points, then re-shuffled till it converged. Afterwards, it did one full loop around all the points to double-check its work. It did this as the larger the dataset, the less likely it is to fit into your cache. By only using the first 100k, it made sure all these points were in its cache, showcasing a huge decline in convergence time.

Iterative prompts were quite effective at improving K-Means performance, and after a few iterations, the output surpassed most students. Although there was a lot of commonality between outputs, they all took their own paths. Simpler strategies like lock tables were effective for small datasets, yet did not scale efficiently with larger datasets. It's interesting to note that these LLMs also took a holistic approach when trying to speed up the performance, and not just an optimization of the for-loops in K-Means. Showcasing that LLMs can generate quite sophisticated code, although it is clear that carefully optimized code from a student can achieve better results in some situations.

Cuckoo

Among the Cuckoo hashing implementations, the Gemini Starter code scaled best overall as seen in **Figure 8**. This was primarily due to its strategy of resizing the hash table early, which reduced contention and allowed inserts to proceed with less interference. However, this came at the cost of performance: frequent resizing introduced significant overhead, making it slower in terms of operations per second compared to more optimized implementations, as seen in **Figure 9**. Interestingly, the Claude Starter implementation revealed a surprising behavior—when an insertion failed, it would overwrite a randomly chosen slot in the table, potentially discarding existing entries without resolution. **Figure 10** makes it clear that the single fastest implementation at 10 M operations on 16 threads was in fact the sequential solution from classmate B, while the best-performing parallel code was LLM-generated: the Palmieri-prompted Claude variant.

The Student prompt achieved modest speedups when using only a few threads. However, they still suffered a contention wall as thread counts rose. The Palmieri prompt generated code with eviction policies and proactive resizing; this led to great scaling now within the ChatGPT-generated code. The Palmieri prompt's Gemini and Claude generated code also scaled well, but at the expense of some microseconds under light loads due to extra conditional logic.

The Emir-prompted implementation delivered the lowest per-thread latency of all the LLM-generated variants and scaled almost linearly up to eight threads. However, at 16 threads its performance leveled off, revealing that contention on shared data structures eventually throttled any further speedup.

Moreover, **Figure 11** shows that most implementations scale almost linearly from 1 K up to 10 M operations, but two classmate submissions (classmate B's parallel variant and classmate G) exhibit even shallower slopes—hinting that at a still larger scale (e.g., a billion operations) they might overtake both the AI-generated parallel and the sequential baselines.

RAG

In general, RAG performance was not great. Most implementations performed as well or worse than pure prompting. Nonetheless, there were some interesting outputs for both Bank and Cuckoo.

Bank

In many implementations, the RAG precomputed the operation (deposit or balance) before timing the program. This was a surprising optimization to see because it is not initially intuitive. More optimizations included using compiler hints, including vectorization, hardware transactional memory, using thread local storage, padding mutexes and locks to the cache line, and shrinking the cache line to be as small as possible.

However, the most interesting LLM output was providing a "future ideas & trade-offs" section. In this output, the LLM suggested "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." When asked to implement the future ideas, the LLM was not able to, and made a slower version. Nevertheless, this was incredibly interesting as it provided a way for an experienced programmer to think about potential additional optimizations.

Cuckoo

In cuckoo, many of the implementations used prime numbers for hashing. We also saw implementations using inheritance and stripping. The most interesting result was implementing its own SpinLock, which made use of hardware synchronization and _mm_pause.

Designing a New Procedure

In an attempt to overcome the RAG's shortcomings, we designed a new procedure that would allow the LLMs to have more control. The first step would be to provide the LLM with the prompt and inform it that it would be provided with a problem it would need to solve and that it had a RAG available to use. Once the LLM acknowledged that it understood the process, we would follow up with the project description. At this point, the LLM would think for a few seconds and follow up with a prompt for the RAG. Next, we would query the RAG with this prompt and return its output to the LLM, which the LLM would use to solve the problem.

While this new procedure did not return any better results in comparison to the previous RAG, it did provide consistent results. Retrying this process on the same prompt over and over again returned surprisingly similar implementations with similar runtimes. This was not observed with the previous RAG. If we were looking for consistent results, then this new procedure would be great.

Conclusion

In conclusion, we find that LLMs' capability to generate good, efficient, and scalable parallel and concurrent code is still lacking. More often than not, LLMs produce code that is on par, worse, or marginally better than our classmates' code. In some cases, however, LLMs do produce more efficient code than even the best available student implementations. However, we believe that these results are few and far between and may be attributed to random chance or specific wording in the prompts that hinted at a better output for specific problems.

Our initial thoughts suggested that implementing a RAG would be a great way to increase the performance of LLMs' code generation. However, we find that the additional context did not help the LLMs; in some cases, it may have done the opposite. We believe that the LLM context length is not large enough to accommodate all of the context, project description, and the additional RAG context. Furthermore, it is highly likely that LLMs have already been trained on the content we indexed, further bogging down LLM memory.

Overall, unless those responsible for training LLMs can parse more efficient and scalable parallel and concurrent code, LLMs will not be able to produce results that are significantly better than what a senior college student taking a parallel computing course could write. Instead, LLMs can and should be used as a vehicle to guide the direction of a coder's mind. More specifically, there is potential to create LLM agents that are tuned to specifically provide advice, not code.

Figures

Bank

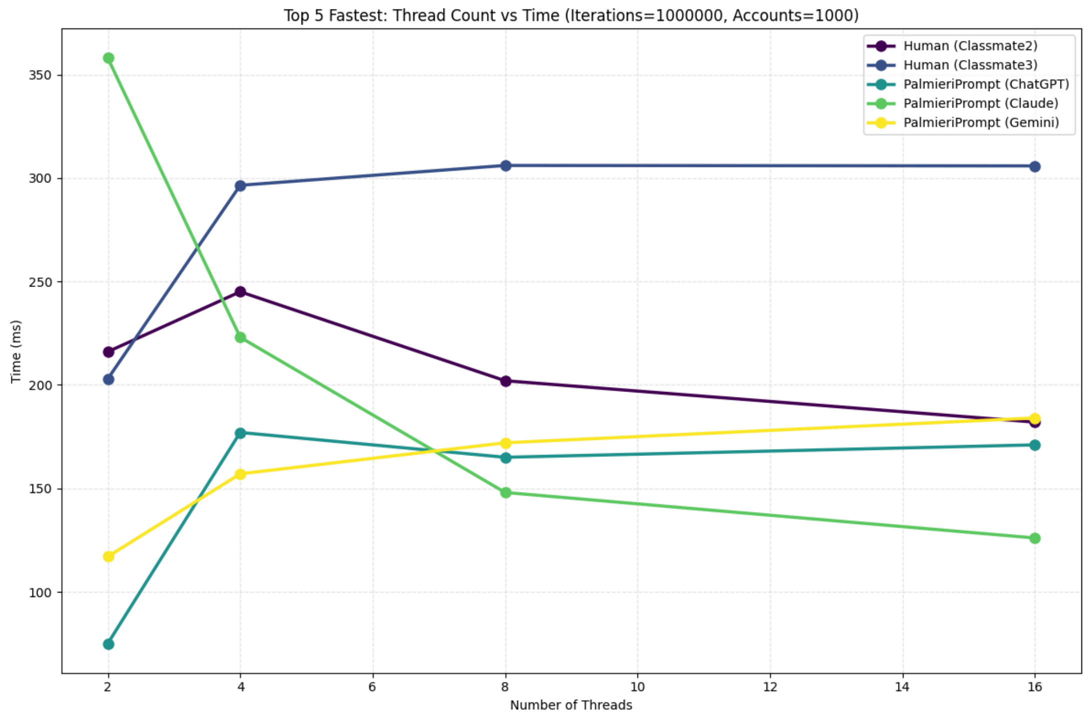

Figure 1

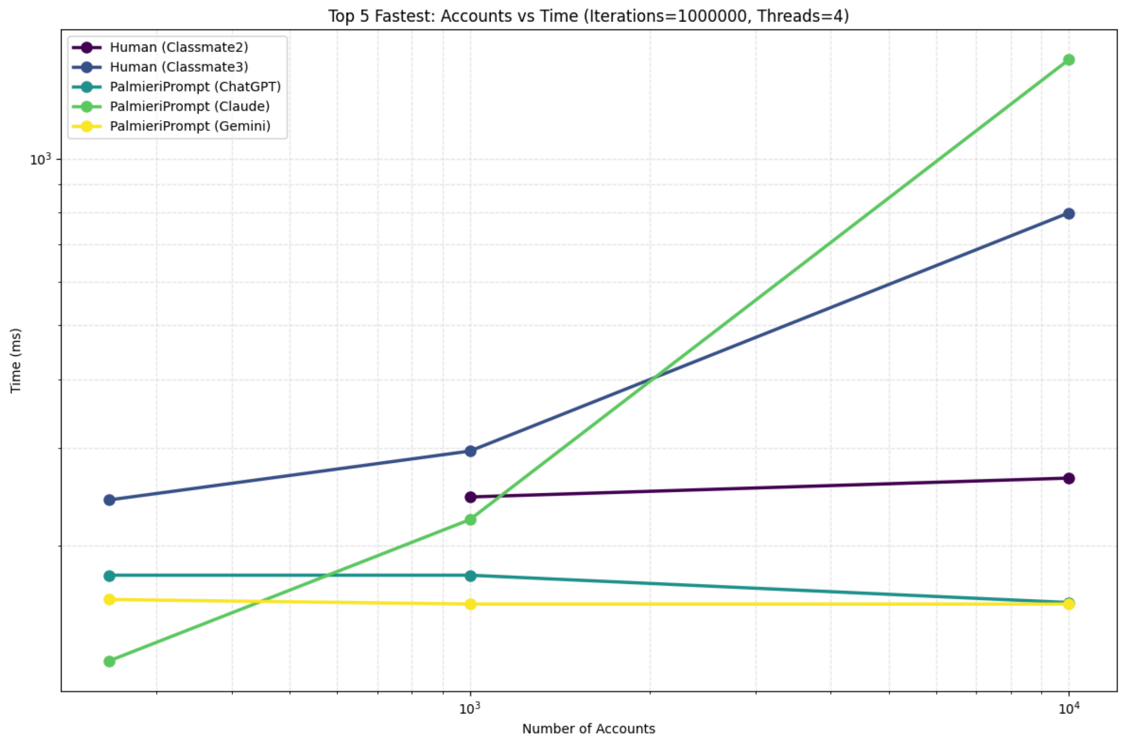

Figure 2

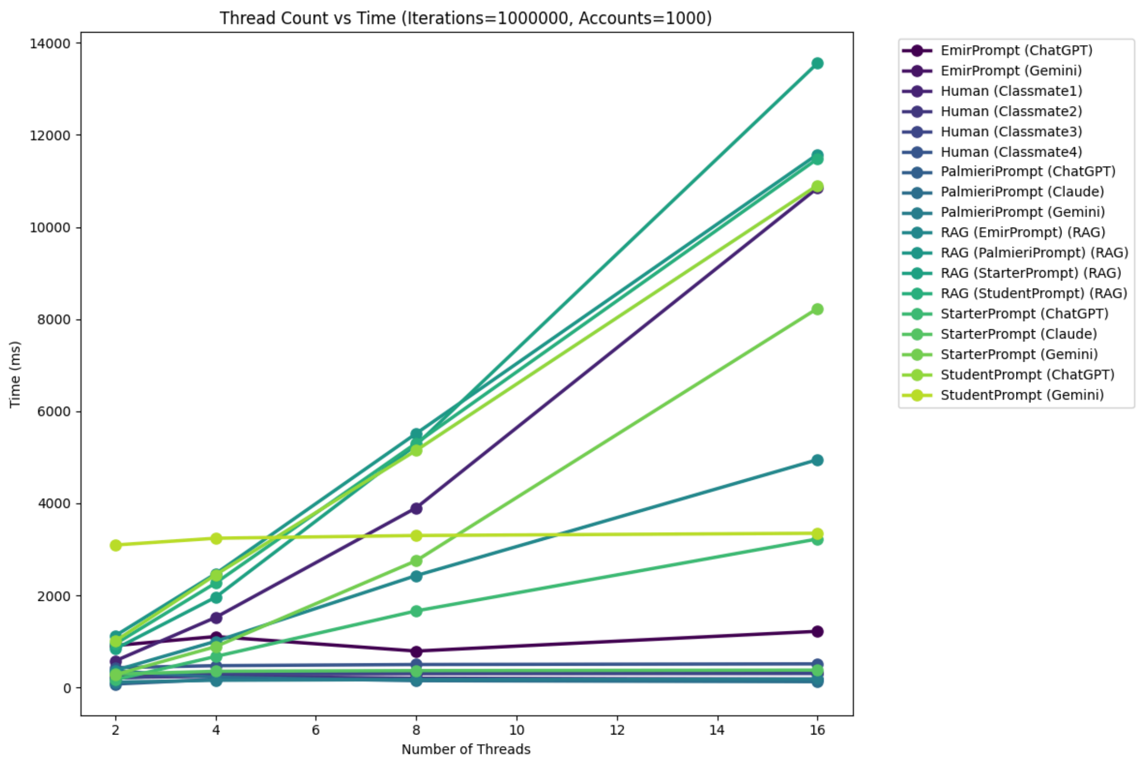

Figure 3

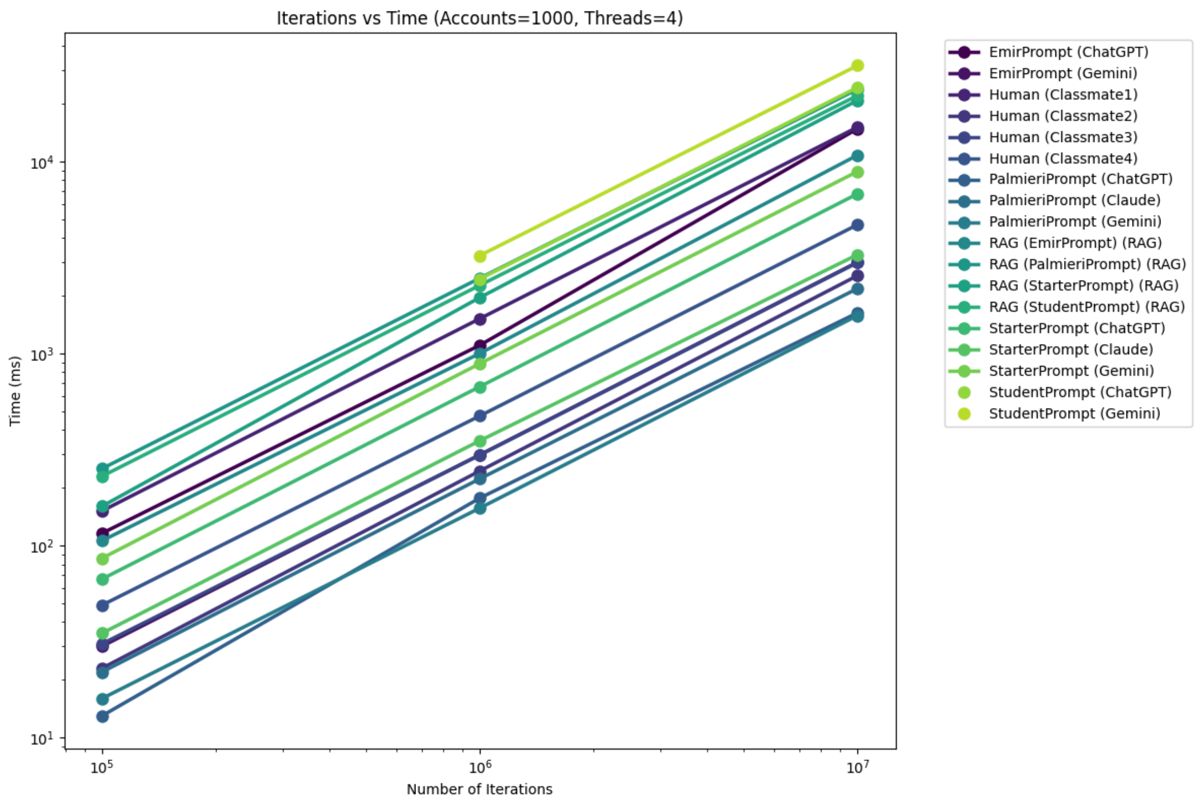

Figure 4

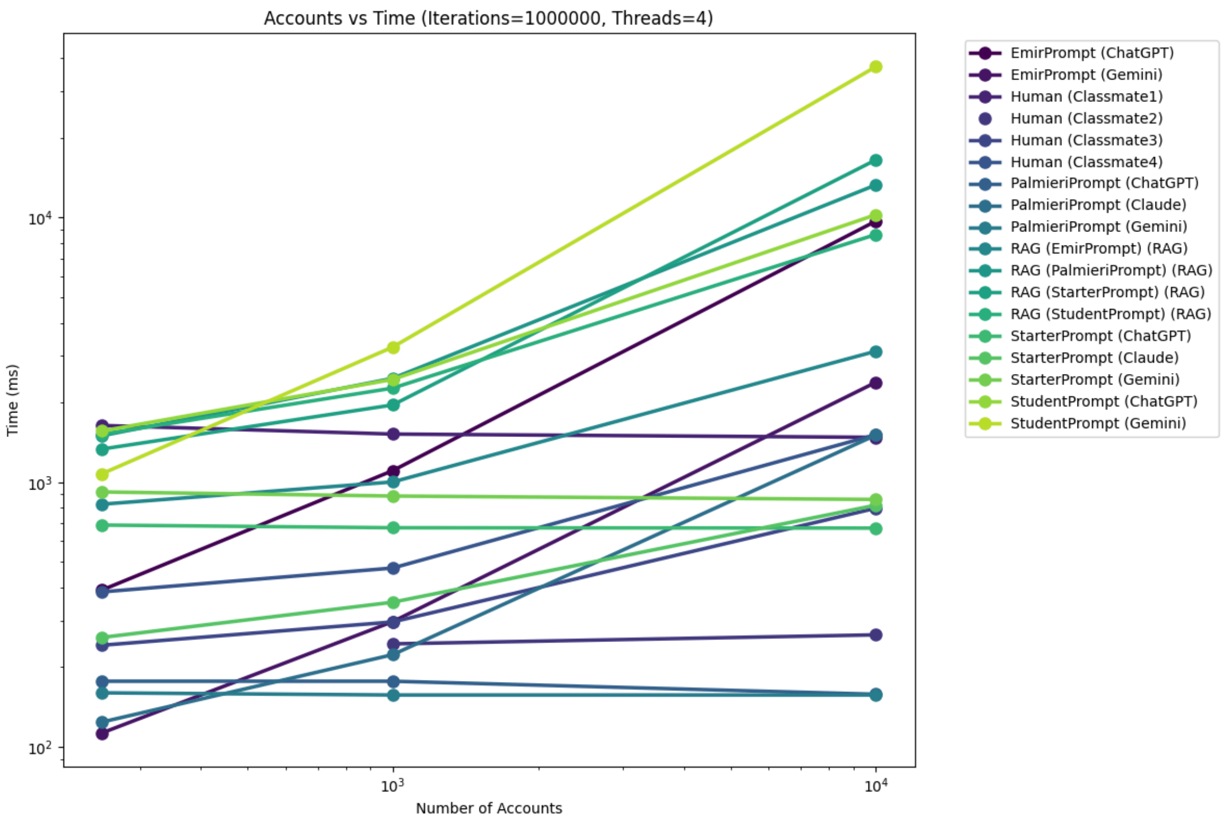
Accounts vs Time (Iterations=1000000, Threads=4)
Time (ms)
Number of Accounts
10^2
10^3
10^4
EmirPrompt (ChatGPT)
EmirPrompt (Gemini)
Human (Classmate1)
Human (Classmate2)
Human (Classmate3)
Human (Classmate4)
PalmieriPrompt (ChatGPT)
PalmieriPrompt (Claude)
PalmieriPrompt (Gemini)
RAG (EmirPrompt) (RAG)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

Figure 5

K-Means:

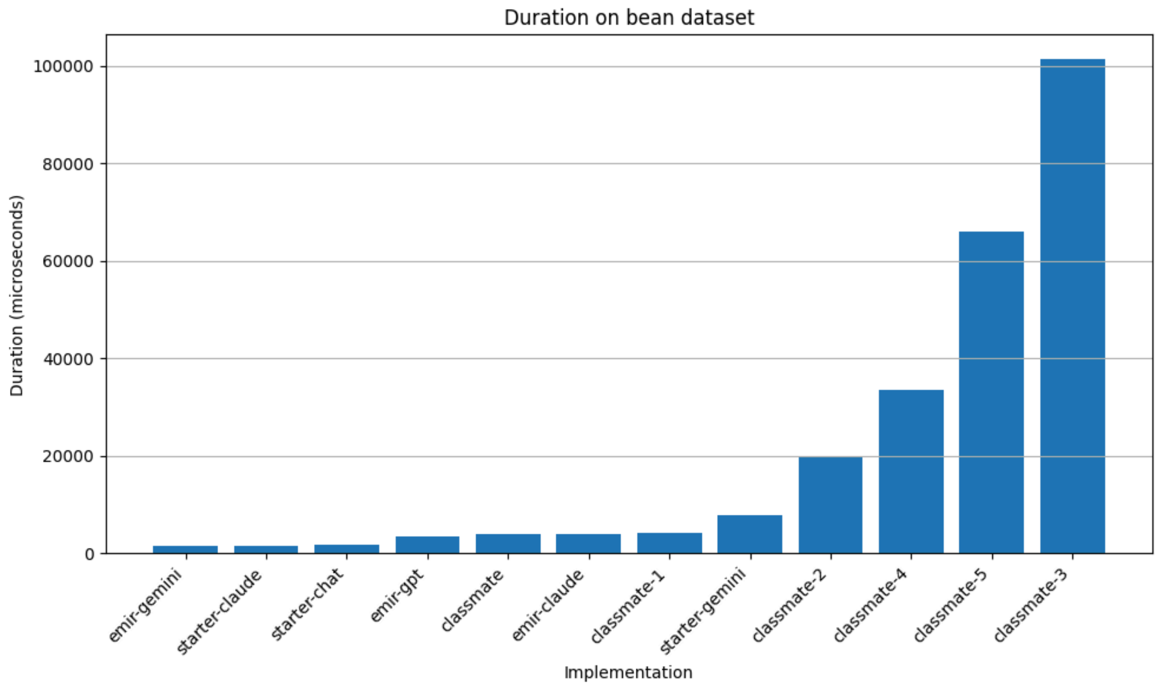

Figure 6

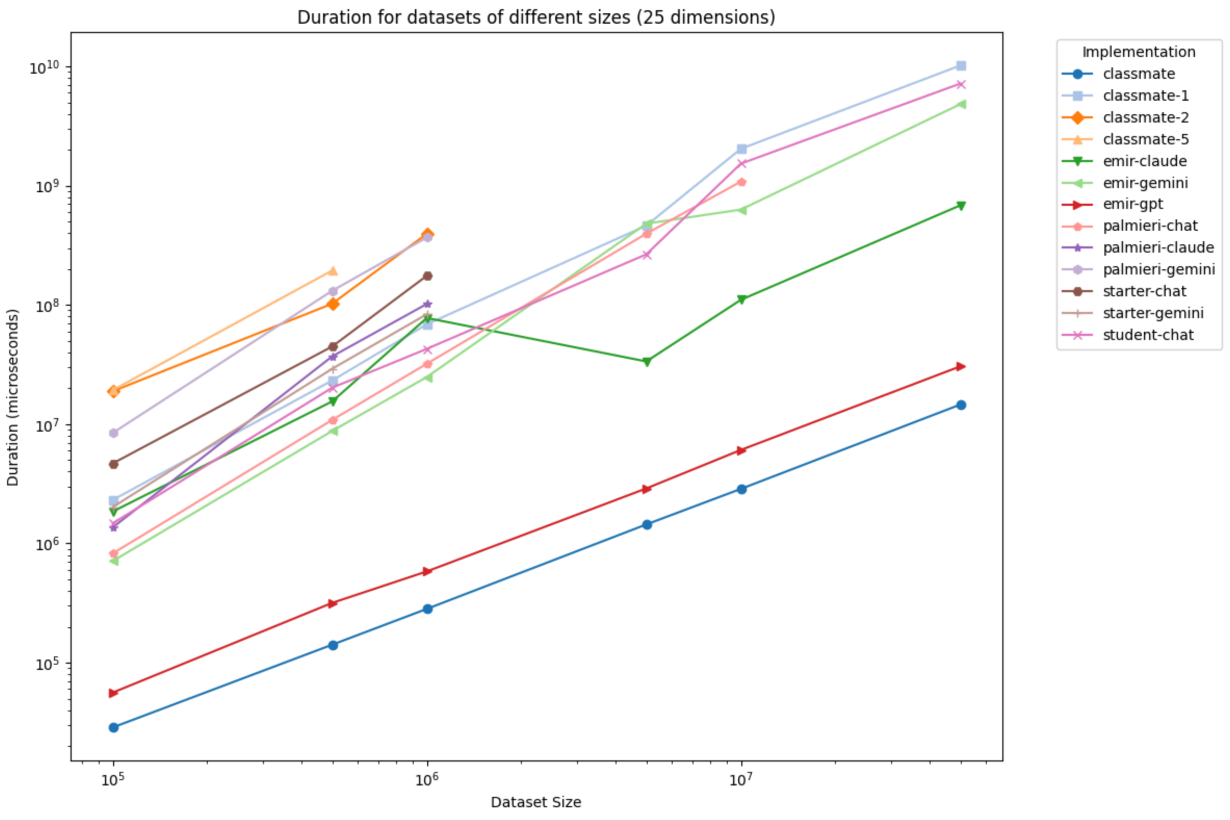

Figure 7

Cuckoo

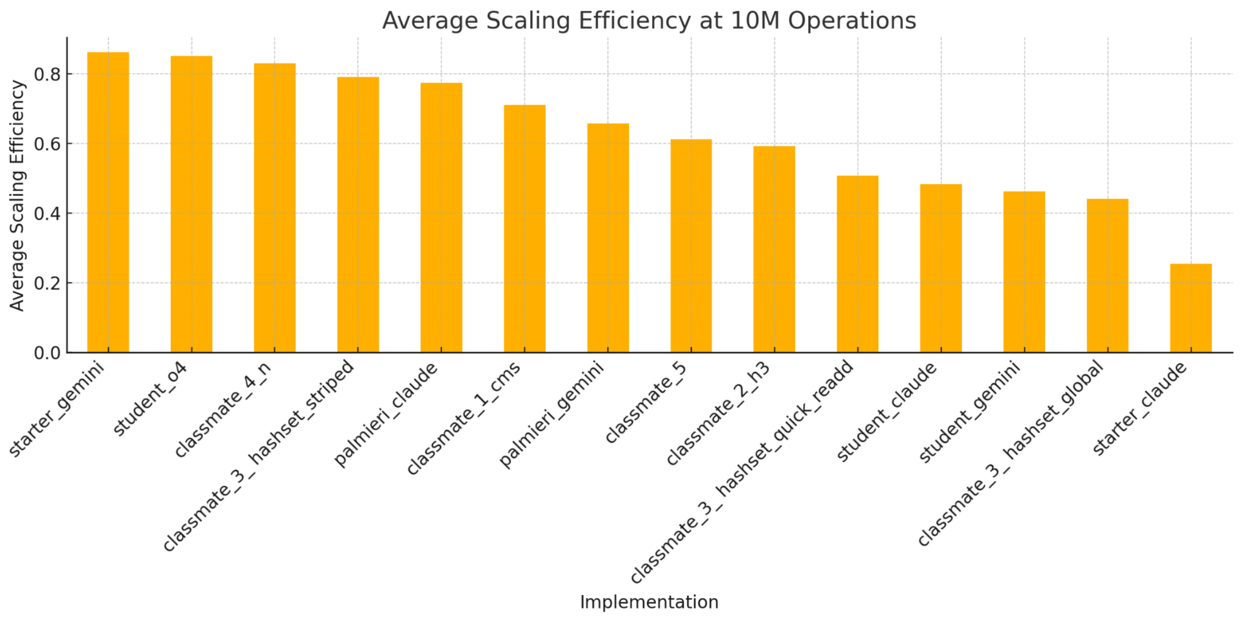

Figure 8

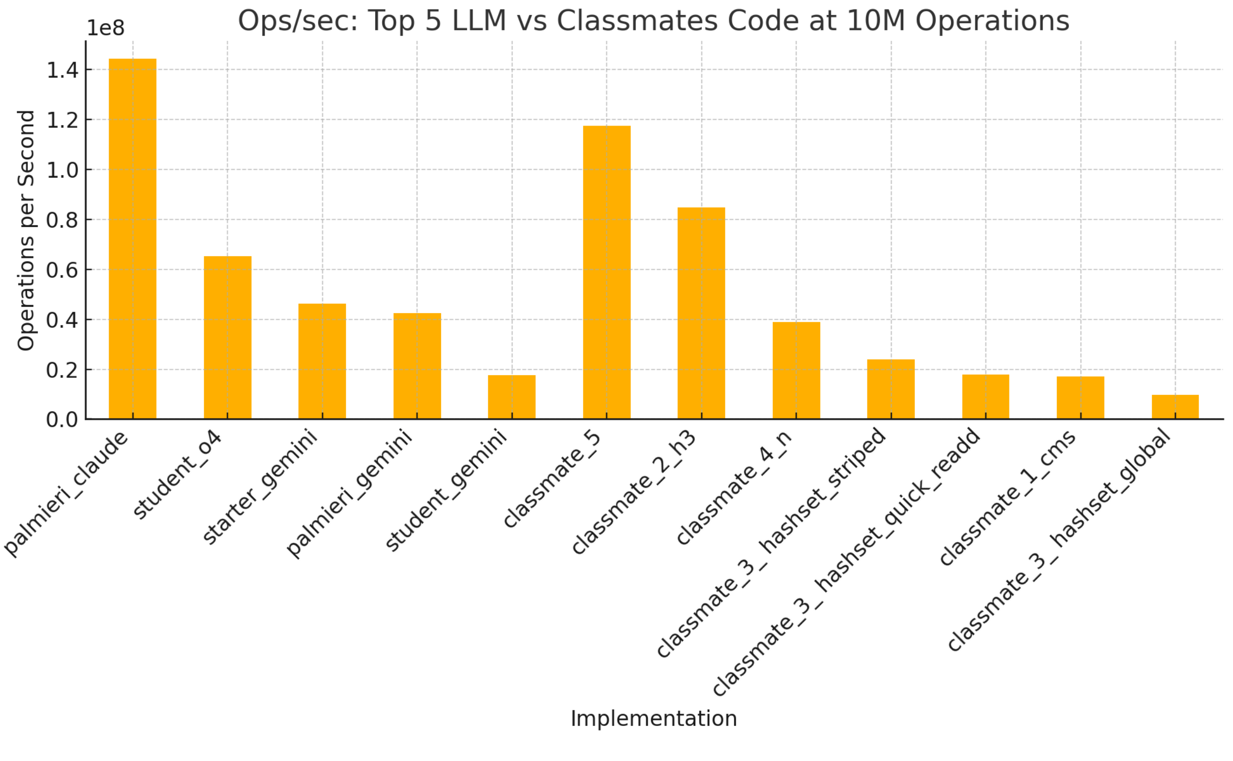

Figure 9

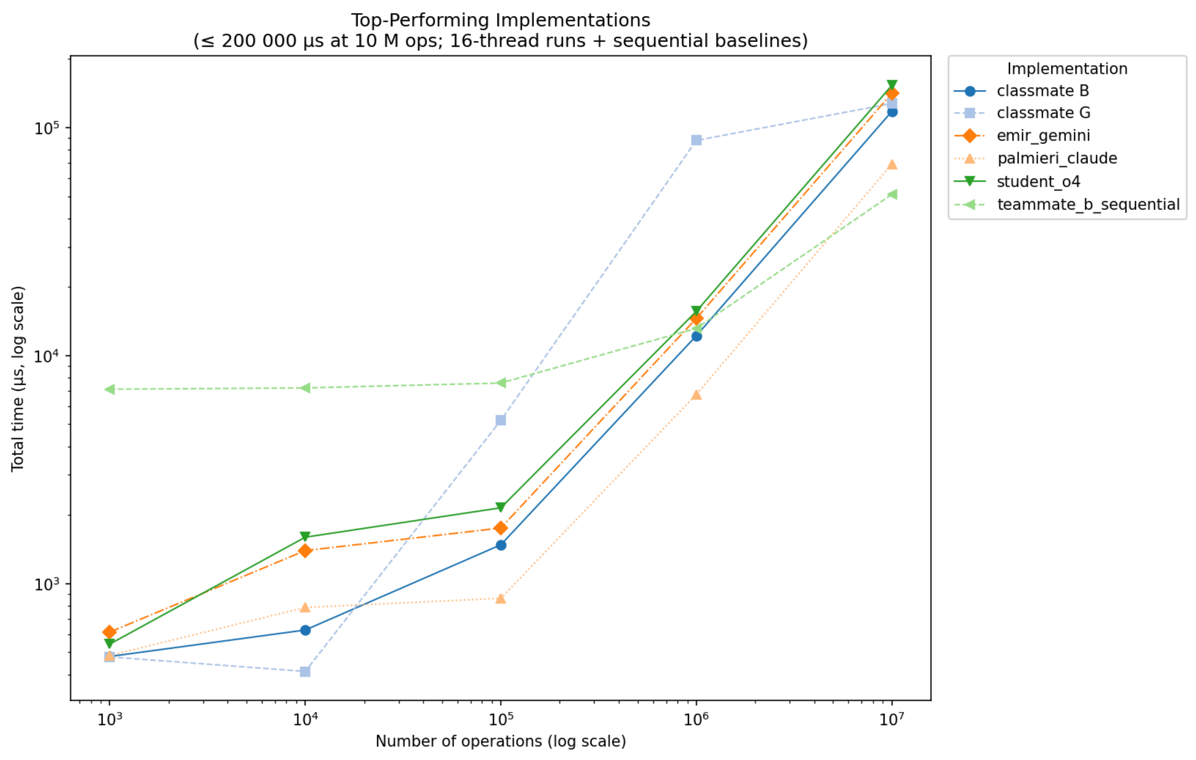

Figure 10

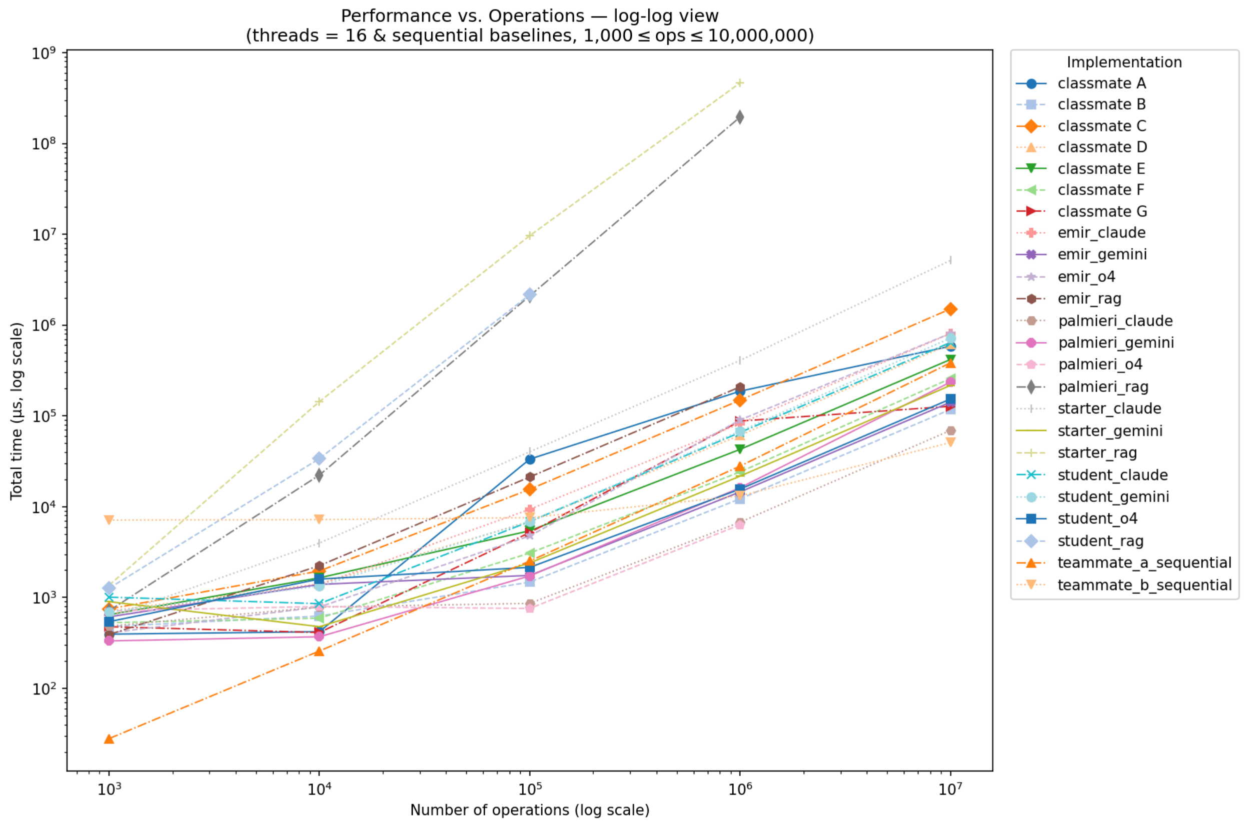

Figure 11