DPS921 Final Report

By: Adam Davis, Agam Singh Talwar, and Meetsimar Kaur

For: Edan Burton

Class: DPS921NSA

Date: 2025-08-14

Table of Contents

Introduction

Can analyzing AI chat responses in parallel be more efficient? This question is increasingly relevant as AI systems are no longer isolated tools, but increasingly active actors within a larger system. Communicating with other AI models, integrating into end user applications, as well as external programs via APIs like MCP (Model Context Protocol). By exploring parallelism in this context, we hope to better understand how concurrency is, and see if these techniques make noticeable performance improvements.

Background

Swift provides a modern concurrency model designed to simplify and improve parallel computing. At its core is structured concurrency, which leverages keywords such as async/await and TaskGroup to coordinate tasks in a way that is both predictable and efficient serving as Swift's high-level alternative to traditional threading. In addition, Swift supports asynchronous streams, such as AsyncThrowingStream, which enable real-time token streaming. This allows developers to handle partial results from a task as they are produced, without waiting for the entire process to complete, making it especially useful for applications like AI chat assistants where responsiveness is critical. Unlike low-level C/C++ threads, which are susceptible to race conditions, and use shared memory. Swift uses a concept known as Task Groups, which utilizes internal queues to prevent race conditions and memory access issues. At the price of some additional overhead at runtime to manage the actor.

Literature Review

Existing research has focused more on advances in large language models (LLMs) such as GPT-4, LLaMA, and Grok. These models offer text generation, summarization, and reasoning capabilities (Lattner, C., & Adve, V). In particular, LLaMA, is recognized for being both compact and competitive, outperforming GPT-3 on benchmarks despite having fewer

parameters. For this reson we opted to use TinyLama as the default model for the experiment. Combined with tools like Ollama any user with sufficient hardware capabilities can run one of these models locally on their device (Ollama Team).

For AI powered applications performance is critical, so that user experience remains high and models gain the resources needed. Apple enables this via structured concurrency in Swift, using features like async/await and Task Groups (Swift Project). These features allow for the safe management of parallel tasks and prevent race conditions from occurring. Task Groups, in particular, facilitate the of multiple asynchronous subtasks in parallel which is ideal for the use case in the experiment. As it allows for a large problem is to be divided into components, processed concurrently, and then recomposed at the end.

Interestingy, local AI tools are gaining traction for the added privacy and convenience they provide. According to a recent article, local AI lets users retain complete control over their data, work offline, and avoid ongoing costs associated with cloud-based services like ChatGPT or Copilot (Devine). While most literature focuses on model performance which is critical. There is a gap in understanding local AI responses and analysing, which we hope to investigate with our experiment.

System Design

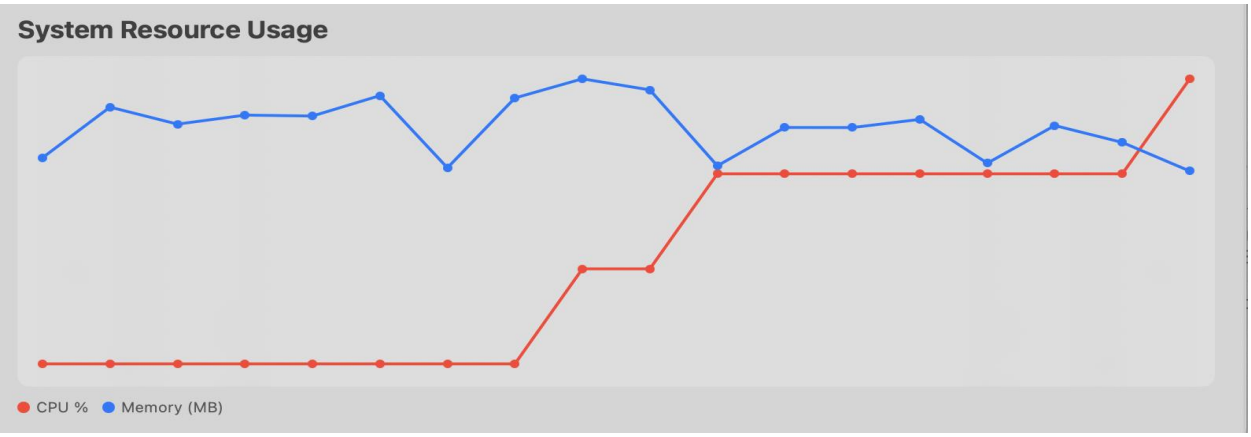
The system was built using Swift 5.9 as the primary programming language, chosen for its modern concurrency features as well as good performance. The user interface was developed with SwiftUI, which allowed for a declarative and responsive design. On the backend, the system is connected to a local Ollama server to generate AI responses. To efficiently manage concurrency and ensure responsiveness, the system utilized Task Groups. Task Groups are particularly well-suited for problems involving geometric decomposition or parallelizable workloads, as they allow multiple subtasks to run concurrently while still maintaining structured control over execution.

In practice, the AI-generated response was divided into smaller, well-defined sections. Each section was assigned to an individual task within the Task Group, which then performed its computations in parallel. Once all tasks are completed execution, the program collects the results, and it is returned to the user. This design ensures that the application should be able to be performant due to its use of the structured concurrency model provided by Swift.

Experiment

The experiment was conducted on a Macbook Pro, with an M1 Chip and 8 GB of RAM. Using TinyLama model. The same set of 10 prompts were given in both the sequential and parallel versions of the experiment. Each prompt was representative of typical end-user queries, ranging from short factual requests to longer open-ended ones that generate larger outputs.

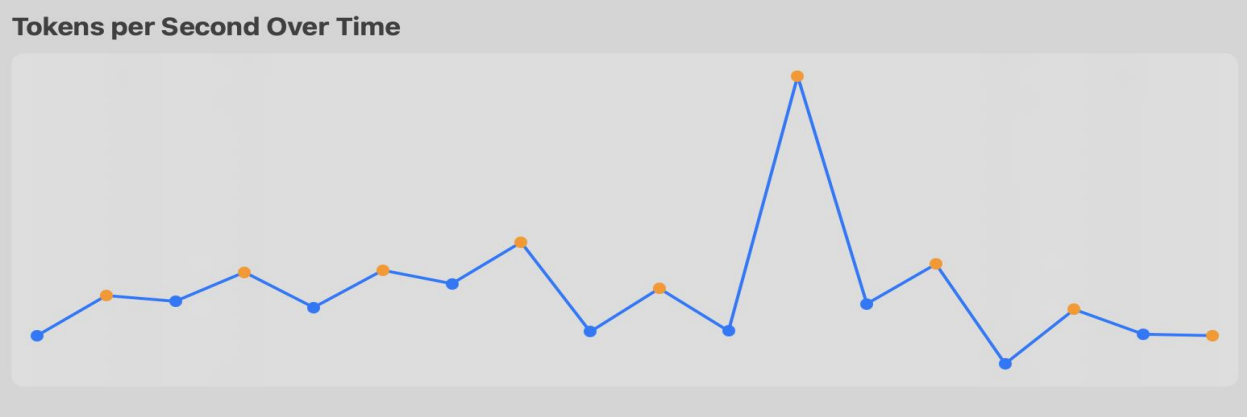
System Resource Usage



From the system monitoring data, there are two measurements CPU usage and memory utilization. Memory usage remained relatively level and high throughout the experiment. This was regardless of whether the sequential or parallel method was used. This suggests that analyzing AI responses is inherently memory-intensive, and the limiting factor for scaling up such applications may often be available RAM rather than raw processing power. With only 8 GB of memory on the test machine, resource pressure was a concern.

In contrast, CPU usage showed a gradual growth pattern it gradually increased over the lifetime of the experiment, particularly in the parallel implementation. This trend suggests that concurrency allows the CPU to be more effectively utilized over time. With multiple tasks keeping the processor cores active rather than letting them idle between sequential operations. Interestingly, this means long-running applications or services that handle a steady flow of requests could gain significant benefits from concurrency, as CPU resources are distributed more efficiently. Short-lived tasks, however, may not fully realize these benefits since the overhead of managing concurrency can outweigh its advantages.

Tokens Analyzed per Second



In the above graph orange dots represent parallel analysis, and blue represents sequential analysis. This metric provides insight into how quickly the system can consume and process AI generated responses. In this graph a higher value is better as it means more tokens are created in the same amount of time. The overall trend shows that regardless of the response, the parallel method tends to be able to consume more tokens than the sequential analysis. This is particularly noticeable on AI responses that are very large, such as the peak of the graph, which analyzed many times greater then the sequential version. However, this benefit is marginal for smaller responses, as the additional overhead of running in parallel mode becomes apparent.

Response Time by Mode

This is likely the most important data collected from the procedure. Response time is a useful measure of the end user experience. In this graph the blue bars represent sequential mode, while the orange bars represent parallel mode. Lower response time being an indicator of better user experience. Across nearly all test cases, the parallel implementation returned results faster than the sequential approach. The difference was particularly noticeable for medium-to-large responses, where concurrency reduced response times by several seconds. The only exception is the very short responses. Like in the right most test on the graph sequential version outperformed the parallel version. This is likely due to the overhead of task management in the Swift runtime. Which is only noticeable with the small workload. However, for medium or large responses the parallel solution remains the best for user experience.

Limitations

While the experiment yielded valuable insights, it is important to acknowledge several limitations that restrict the scope and generalizability of the findings.

Model and Device Constraints

The analysis was conducted exclusively with a single AI model, TinyLama, on a single hardware platform, the Apple M2 Mac. As a result, the outcomes may not accurately

reflect performance or behavior across other models, architectures, or devices with different computational capabilities. A more diverse set of models and hardware would provide a stronger basis for comparison and broader applicability.

Sample Size of Prompts

The study relied on a relatively small set of 10 prompts. Although these were intentionally chosen to cover a range of scenarios, the sample size remains limited and may not fully capture the variability in model performance. A larger and more diverse dataset of prompts would help reveal more nuanced patterns in response length, quality, and consistency.

Narrow Evaluation Metrics

The experiment primarily focused on user experience and performance benchmarks such as response length and timing. Other critical aspects, including energy efficiency, computational cost, and subjective user satisfaction, were not considered. These dimensions could significantly influence the real-world utility of AI systems and should be incorporated in future evaluations.

Contextual Limitations

The prompts used in the study were generated in a controlled experimental setting, which may not fully replicate real-world usage. In practical applications, user inputs can be noisier, more ambiguous, or domain-specific. This context gap limits how directly the results can be applied to real-world AI deployments.

Future Work

Should we be given the opportunity to continue with this project. There are several improvements that will be made. Firstly, we would like to rerun the experiment using several different AI models instead of a single one. Including performance optimized ones like TinyLlama to large scale models like Gemni. Secondly, we would like to test with different hardware configurations to see how that would change the results. For example,

testing on a CPU with more cores to see how that would affect the parallelization, or how the amount of RAM affects the speed of the model. Finally, we would also like to add a feature for users to rank the responses so that it can be included within the analysis.

Conclusion

This project set out to explore whether analyzing AI responses in parallel could improve efficiency compared to a traditional sequential approach. The results demonstrate that parallel execution in Swift does indeed provide measurable benefits, particularly in reducing response time and increasing throughput for medium to large AI-generated outputs. By leveraging Swift's modern concurrency features, such as Task Groups, the system was able to utilize CPU resources more effectively, keeping processor cores engaged and minimizing idle time.

At the same time, the findings highlight that parallelism is not universally beneficial. For very short responses, the overhead of managing concurrent tasks outweighed the gains, making sequential execution slightly faster. This suggests that while concurrency can significantly improve user experience in scenarios involving larger workloads, it may not always be the optimal choice for lightweight tasks.

Beyond raw performance, the experiment also underscored some broader implications. Memory usage remained consistently high, pointing to RAM as a limiting factor in scaling AI analysis systems. This emphasizes that efficiency improvements from concurrency must be considered alongside hardware constraints. Furthermore, while our study focused on measurable performance metrics, other important factors, such as energy efficiency and user satisfaction, remain areas for future exploration.

In conclusion, our experiment supports the claim that parallel analysis of AI responses offers clear advantages in responsiveness and efficiency under the right conditions. While the study is constrained by its limited scope using a single model, device, and small prompt set it provides a valuable foundation for understanding how concurrency can be applied in real-world AI applications. With broader testing across models, hardware

platforms, and evaluation metrics, future research could further validate and expand upon these findings, ultimately contributing to the design of more scalable, efficient, and user-friendly AI systems.

References

Devine, Richard. “ChatGPT Isn’t the Only Game in Town—Here’s Why Local AI Might Be Better (and How to Use It on Your PC).” Windows Central, 9 Aug. 2025, www.windowscentral.com/artificial-intelligence/5-reasons-to-use-local-ai-tools-over-copilot-or-chatgpt-anyone-can-try-it-so-why-wouldnt-you. Accessed 17 Aug. 2025.

Ollama Team. “Ollama/Docs/Api.md at Main · Ollama/Ollama.” *GitHub*, github.com/ollama/ollama/blob/main/docs/api.md.

Swift Project. “Documentation.” *Docs.swift.org*, docs.swift.org/swift-book/documentation/the-swift-programming-language/concurrency/.

Lattner, C., & Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 75–88. IEEE. https://doi.org/10.1109/CGO.2004.1281665

Zhang, Q., & Lin, M. (2022). Evaluating parallel task execution in large language models. *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 3021–3032. Association for Computational Linguistics. https://doi.org/10.18653/v1/2022.emnlp-main.214

Appendix

Raw Test Results

Mode	Time (s)	Tokens/s	CPU %	Memory	Model
Sequential	9.16	43.9	15	540	Tinyllama:latest
Parallel	6.98	59.0	15	580	Tinyllama:latest
Sequential	2.69	56.0	15	560	Tinyllama:latest
Parallel	4.51	67.9	15	570	Tinyllama:latest
Sequential	6.77	54.5	13	600	Tinyllama:latest
Parallel	6.09	65.6	15	640	Tinyllama:latest
Sequential	2.77	79.2	15	590	Tinyllama:latest
Parallel	6.95	45.3	40	610	Tinyllama:latest
Sequential	5.78	61.7	40	580	Tinyllama:latest
Parallel	5.78	61.7	43	560	Tinyllama:latest
Sequential	0.90	45.6	70	510	Tinyllama:latest
Parallel	0.27	142.0	70	520	Tinyllama:latest
Sequential	3.06	55.9	70	530	Tinyllama:latest
Parallel	2.33	70.9	70	550	Tinyllama:latest
Sequential	8.06	33.2	70	570	Tinyllama:latest
Parallel	2.47	53.8	70	600	Tinyllama:latest
Sequential	3.92	44.4	70	620	Tinyllama:latest
Parallel	8.77	43.9	90	500	Tinyllama:latest