|  |  |  |
|---|---|---|
| **Paper Title** | **:** An LLMCompiler for Parallel Function Calling | |
| **Authours** | **:** Sehoon Kim | Suhong Moon |
| | RyanTabrizi | Nicholas Lee |
| | Michael W. Mahoney | Kurt Keutzer |
| | Amir Gholami | |
| **Date of Publication** | **:** 6 Feb 2024 | |
| Name | : Deep Salunkhe | |
| Roll No | : 21102A0014 | |

## Summary

Recent language models (LLMs) have demonstrated remarkable performance on complex reasoning benchmarks. They can execute external function calls to overcome limitations such as knowledge cutoffs, poor arithmetic skills, or lack of access to private data. However, current methods for calling multiple functions require sequential reasoning, leading to high latency, cost, and sometimes inaccurate behavior.

To address these issues, the authors introduce LLMCompiler, which executes functions in parallel to efficiently orchestrate multiple function calling. LLMCompiler streamlines parallel function calling with three components:

(i) an LLM Planner that formulates execution plans
(ii) a Task Fetching Unit that dispatches function calling tasks
(iii) an Executor that executes these tasks in parallel.

LLMCompiler generates an optimized orchestration for function calls and can be used with both open-source and closed-source models. When benchmarked against ReAct, a notable approach that sequentially executes function calls, LLMCompiler demonstrated consistent latency speedup of up to 3.7x, cost savings of up to 6.7x, and accuracy improvement of up to ~9%.

The methodology of LLMCompiler involves the following steps:

1. The LLM Planner generates a Directed Acyclic Graph (DAG) of tasks with inter-dependencies.
2. The Task Fetching Unit dispatches tasks to the Executor in parallel based on their dependencies.
3. The Executor executes the tasks using associated functions.
4. Results (observations) are forwarded back to the Task Fetching Unit to unlock dependent tasks.
5. The final answer is delivered to the user.

LLMCompiler is inspired by classical compilers and optimizes parallel function calling performance. It improves latency, cost, and accuracy by minimizing interference from intermediate function call outputs, addressing the inefficiencies of sequential function calling and repetitive LLM invocations seen in approaches like ReAct.

While related work like Skeleton-of-Thought reduces latency through application-level parallel decoding, it has limitations in handling interdependent tasks. Other methods like Decomposed Prompting, Step-Back Prompting, and Plan-and-Solve Prompting improve accuracy but not latency. In contrast, LLMCompiler enables parallel function calling and dynamic replanning.