**Chapter 3\_ Parallelization**

Chapter 3: Parallelization

**Parallelization Pattern Overview**

In the previous chapters, we've explored Prompt Chaining for sequential workflows and Routing for dynamic decision-making and transitions between different paths. While these patterns are essential, many complex agentic tasks involve multiple sub-tasks that can be executed *simultaneously* rather than one after another. This is where the **Parallelization** pattern becomes crucial.

Parallelization involves executing multiple components, such as LLM calls, tool usages, or even entire sub-agents, concurrently (see Fig.1). Instead of waiting for one step to complete before starting the next, parallel execution allows independent tasks to run at the same time, significantly reducing the overall execution time for tasks that can be broken down into independent parts.

Consider an agent designed to research a topic and summarize its findings. A sequential approach might:

1. Search for Source A.
2. Summarize Source A.
3. Search for Source B.
4. Summarize Source B.
5. Synthesize a final answer from summaries A and B.

A parallel approach could instead:

1. Search for Source A *and* Search for Source B simultaneously.
2. Once both searches are complete, Summarize Source A *and* Summarize Source B simultaneously.
3. Synthesize a final answer from summaries A and B (this step is typically sequential, waiting for the parallel steps to finish).

The core idea is to identify parts of the workflow that do not depend on the output of other parts and execute them in parallel. This is particularly effective when dealing with external services (like APIs or databases) that have latency, as you can issue multiple requests concurrently.

Implementing parallelization often requires frameworks that support asynchronous execution or multi-threading/multi-processing. Modern agentic frameworks are designed with asynchronous operations in mind, allowing you to easily define steps that can run in parallel.

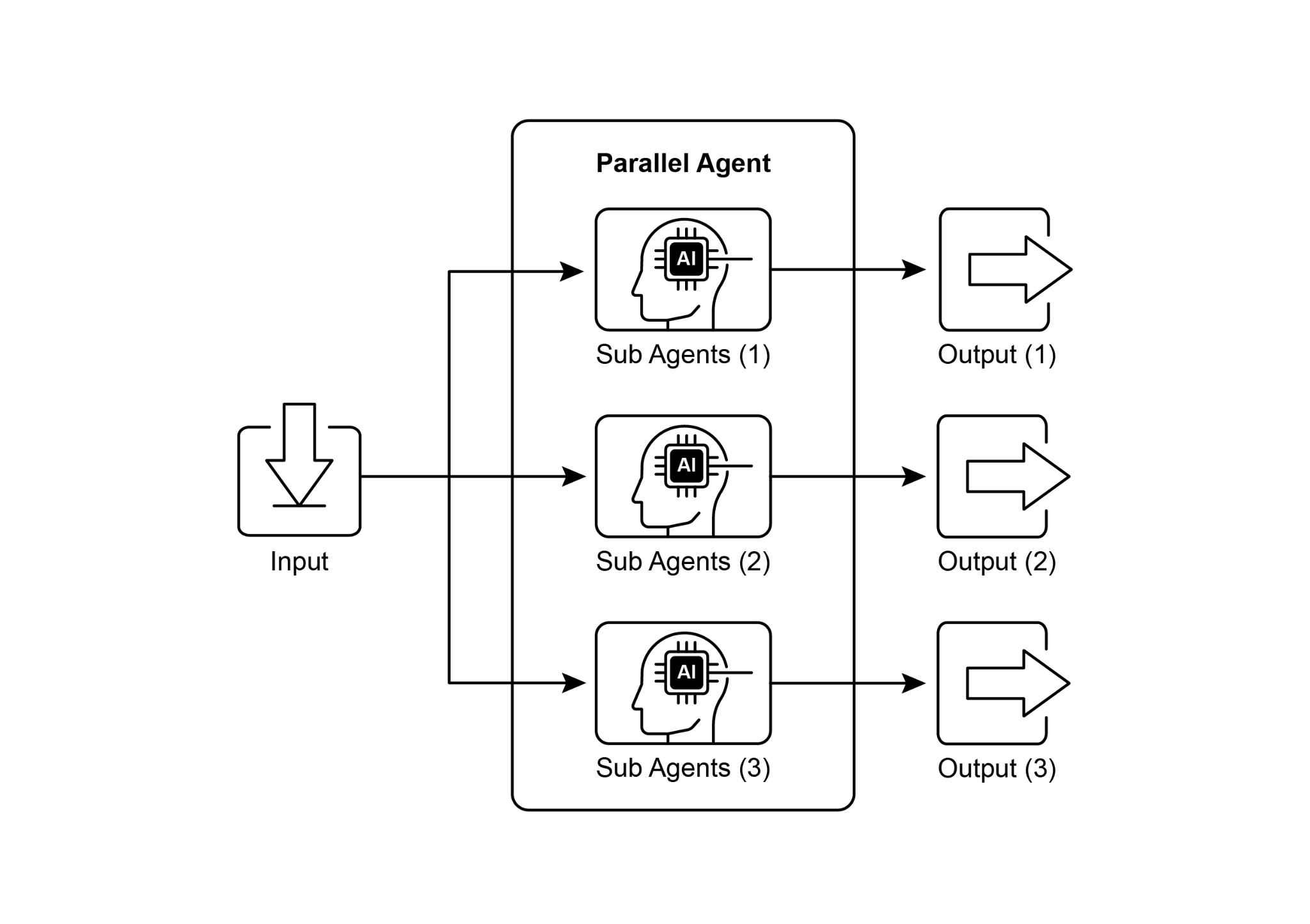


Fig.1. Example of parallelization with sub-agents

Frameworks like LangChain, LangGraph, and Google ADK provide mechanisms for parallel execution. In LangChain Expression Language (LCEL), you can achieve parallel execution by combining runnable objects using operators like | (for sequential) and by structuring your chains or graphs to have branches that execute concurrently. LangGraph, with its graph structure, allows you to define multiple nodes that can be executed from a single state transition, effectively enabling parallel branches in the workflow. Google ADK provides robust, native mechanisms to facilitate and manage the parallel execution of agents, significantly enhancing the efficiency and scalability of complex, multi-agent systems. This inherent capability within the ADK framework allows developers to design and implement solutions where multiple agents can operate concurrently, rather than sequentially.

The Parallelization pattern is vital for improving the efficiency and responsiveness of agentic systems, especially when dealing with tasks that involve multiple independent lookups, computations, or interactions with external services. It's a key technique for optimizing the performance of complex agent workflows.

**Practical Applications & Use Cases**

Parallelization is a powerful pattern for optimizing agent performance across various applications:

1. Information Gathering and Research:

Collecting information from multiple sources simultaneously is a classic use case.

* **Use Case:** An agent researching a company.
* **Parallel Tasks:** Search news articles, pull stock data, check social media mentions, and query a company database, all at the same time.
* **Benefit:** Gathers a comprehensive view much faster than sequential lookups.

2. Data Processing and Analysis:

Applying different analysis techniques or processing different data segments concurrently.

* **Use Case:** An agent analyzing customer feedback.
* **Parallel Tasks:** Run sentiment analysis, extract keywords, categorize feedback, and identify urgent issues simultaneously across a batch of feedback entries.
* **Benefit:** Provides a multi-faceted analysis quickly.

3. Multi-API or Tool Interaction:

Calling multiple independent APIs or tools to gather different types of information or perform different actions.

* **Use Case:** A travel planning agent.
* **Parallel Tasks:** Check flight prices, search for hotel availability, look up local events, and find restaurant recommendations concurrently.
* **Benefit:** Presents a complete travel plan faster.

4. Content Generation with Multiple Components:

Generating different parts of a complex piece of content in parallel.

* **Use Case:** An agent creating a marketing email.
* **Parallel Tasks:** Generate a subject line, draft the email body, find a relevant image, and create a call-to-action button text simultaneously.
* **Benefit:** Assembles the final email more efficiently.

5. Validation and Verification:

Performing multiple independent checks or validations concurrently.

* **Use Case:** An agent verifying user input.
* **Parallel Tasks:** Check email format, validate phone number, verify address against a database, and check for profanity simultaneously.
* **Benefit:** Provides faster feedback on input validity.

6. Multi-Modal Processing:

Processing different modalities (text, image, audio) of the same input concurrently.

* **Use Case:** An agent analyzing a social media post with text and an image.
* **Parallel Tasks:** Analyze the text for sentiment and keywords *and* analyze the image for objects and scene description simultaneously.
* **Benefit:** Integrates insights from different modalities more quickly.

7. A/B Testing or Multiple Options Generation:

Generating multiple variations of a response or output in parallel to select the best one.

* **Use Case:** An agent generating different creative text options.
* **Parallel Tasks:** Generate three different headlines for an article simultaneously using slightly different prompts or models.
* **Benefit:** Allows for quick comparison and selection of the best option.

Parallelization is a fundamental optimization technique in agentic design, allowing developers to build more performant and responsive applications by leveraging concurrent execution for independent tasks.

**Hands-On Code Example (LangChain)**

Parallel execution within the LangChain framework is facilitated by the LangChain Expression Language (LCEL). The primary method involves structuring multiple runnable components within a dictionary or list construct. When this collection is passed as input to a subsequent component in the chain, the LCEL runtime executes the contained runnables concurrently.

In the context of LangGraph, this principle is applied to the graph's topology. Parallel workflows are defined by architecting the graph such that multiple nodes, lacking direct sequential dependencies, can be initiated from a single common node. These parallel pathways execute independently before their results can be aggregated at a subsequent convergence point in the graph.

The following implementation demonstrates a parallel processing workflow constructed with the LangChain framework. This workflow is designed to execute two independent operations concurrently in response to a single user query. These parallel processes are instantiated as distinct chains or functions, and their respective outputs are subsequently aggregated into a unified result.

The prerequisites for this implementation include the installation of the requisite Python packages, such as langchain, langchain-community, and a model provider library like langchain-openai. Furthermore, a valid API key for the chosen language model must be configured in the local environment for authentication.

|  |
| --- |
| import os  import asyncio  from typing import Optional  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  from langchain\_core.output\_parsers import StrOutputParser  from langchain\_core.runnables import Runnable, RunnableParallel, RunnablePassthrough  # --- Configuration ---  # Ensure your API key environment variable is set (e.g., OPENAI\_API\_KEY)  try:  llm: Optional[ChatOpenAI] = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)    except Exception as e:  print(f"Error initializing language model: {e}")  llm = None  # --- Define Independent Chains ---  # These three chains represent distinct tasks that can be executed in parallel.  summarize\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("system", "Summarize the following topic concisely:"),  ("user", "{topic}")  ])  | llm  | StrOutputParser()  )  questions\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("system", "Generate three interesting questions about the following topic:"),  ("user", "{topic}")  ])  | llm  | StrOutputParser()  )  terms\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("system", "Identify 5-10 key terms from the following topic, separated by commas:"),  ("user", "{topic}")  ])  | llm  | StrOutputParser()  )  # --- Build the Parallel + Synthesis Chain ---  # 1. Define the block of tasks to run in parallel. The results of these,  # along with the original topic, will be fed into the next step.  map\_chain = RunnableParallel(  {  "summary": summarize\_chain,  "questions": questions\_chain,  "key\_terms": terms\_chain,  "topic": RunnablePassthrough(), # Pass the original topic through  }  )  # 2. Define the final synthesis prompt which will combine the parallel results.  synthesis\_prompt = ChatPromptTemplate.from\_messages([  ("system", """Based on the following information:  Summary: {summary}  Related Questions: {questions}  Key Terms: {key\_terms}  Synthesize a comprehensive answer."""),  ("user", "Original topic: {topic}")  ])  # 3. Construct the full chain by piping the parallel results directly  # into the synthesis prompt, followed by the LLM and output parser.  full\_parallel\_chain = map\_chain | synthesis\_prompt | llm | StrOutputParser()  # --- Run the Chain ---  async def run\_parallel\_example(topic: str) -> None:  """  Asynchronously invokes the parallel processing chain with a specific topic  and prints the synthesized result.  Args:  topic: The input topic to be processed by the LangChain chains.  """  if not llm:  print("LLM not initialized. Cannot run example.")  return  print(f"\n--- Running Parallel LangChain Example for Topic: '{topic}' ---")  try:  # The input to `ainvoke` is the single 'topic' string,  # then passed to each runnable in the `map\_chain`.  response = await full\_parallel\_chain.ainvoke(topic)  print("\n--- Final Response ---")  print(response)  except Exception as e:  print(f"\nAn error occurred during chain execution: {e}")  if \_\_name\_\_ == "\_\_main\_\_":  test\_topic = "The history of space exploration"  # In Python 3.7+, asyncio.run is the standard way to run an async function.  asyncio.run(run\_parallel\_example(test\_topic)) |

The provided Python code implements a LangChain application designed for processing a given topic efficiently by leveraging parallel execution. Note that asyncio provides concurrency, not parallelism. It achieves this on a single thread by using an event loop that intelligently switches between tasks when one is idle (e.g., waiting for a network request). This creates the effect of multiple tasks progressing at once, but the code itself is still being executed by only one thread, constrained by Python's Global Interpreter Lock (GIL).

The code begins by importing essential modules from langchain\_openai and langchain\_core, including components for language models, prompts, output parsing, and runnable structures. The code attempts to initialize a ChatOpenAI instance, specifically using the "gpt-4o-mini" model, with a specified temperature for controlling creativity. A try-except block is used for robustness during the language model initialization. Three independent LangChain "chains" are then defined, each designed to perform a distinct task on the input topic. The first chain is for summarizing the topic concisely, using a system message and a user message containing the topic placeholder. The second chain is configured to generate three interesting questions related to the topic. The third chain is set up to identify between 5 and 10 key terms from the input topic, requesting them to be comma-separated. Each of these independent chains consists of a ChatPromptTemplate tailored to its specific task, followed by the initialized language model and a StrOutputParser to format the output as a string.

A RunnableParallel block is then constructed to bundle these three chains, allowing them to execute simultaneously. This parallel runnable also includes a RunnablePassthrough to ensure the original input topic is available for subsequent steps. A separate ChatPromptTemplate is defined for the final synthesis step, taking the summary, questions, key terms, and the original topic as input to generate a comprehensive answer. The full end-to-end processing chain, named full\_parallel\_chain, is created by sequencing the map\_chain (the parallel block) into the synthesis prompt, followed by the language model and the output parser. An asynchronous function run\_parallel\_example is provided to demonstrate how to invoke this full\_parallel\_chain. This function takes the topic as input and uses invoke to run the asynchronous chain. Finally, the standard Python if \_\_name\_\_ == "\_\_main\_\_": block shows how to execute the run\_parallel\_example with a sample topic, in this case, "The history of space exploration", using asyncio.run to manage the asynchronous execution.

In essence, this code sets up a workflow where multiple LLM calls (for summarizing, questions, and terms) happen at the same time for a given topic, and their results are then combined by a final LLM call. This showcases the core idea of parallelization in an agentic workflow using LangChain.

**Hands-On Code Example (Google ADK)**

Okay, let's now turn our attention to a concrete example illustrating these concepts within the Google ADK framework. We'll examine how the ADK primitives, such as ParallelAgent and SequentialAgent, can be applied to build an agent flow that leverages concurrent execution for improved efficiency.

|  |
| --- |
| from google.adk.agents import LlmAgent, ParallelAgent, SequentialAgent  from google.adk.tools import google\_search  GEMINI\_MODEL="gemini-2.0-flash"  # --- 1. Define Researcher Sub-Agents (to run in parallel) ---  # Researcher 1: Renewable Energy  researcher\_agent\_1 = LlmAgent(  name="RenewableEnergyResearcher",  model=GEMINI\_MODEL,  instruction="""You are an AI Research Assistant specializing in energy.  Research the latest advancements in 'renewable energy sources'.  Use the Google Search tool provided.  Summarize your key findings concisely (1-2 sentences).  Output \*only\* the summary.  """,  description="Researches renewable energy sources.",  tools=[google\_search],  # Store result in state for the merger agent  output\_key="renewable\_energy\_result"  )  # Researcher 2: Electric Vehicles  researcher\_agent\_2 = LlmAgent(  name="EVResearcher",  model=GEMINI\_MODEL,  instruction="""You are an AI Research Assistant specializing in transportation.  Research the latest developments in 'electric vehicle technology'.  Use the Google Search tool provided.  Summarize your key findings concisely (1-2 sentences).  Output \*only\* the summary.  """,  description="Researches electric vehicle technology.",  tools=[google\_search],  # Store result in state for the merger agent  output\_key="ev\_technology\_result"  )  # Researcher 3: Carbon Capture  researcher\_agent\_3 = LlmAgent(  name="CarbonCaptureResearcher",  model=GEMINI\_MODEL,  instruction="""You are an AI Research Assistant specializing in climate solutions.  Research the current state of 'carbon capture methods'.  Use the Google Search tool provided.  Summarize your key findings concisely (1-2 sentences).  Output \*only\* the summary.  """,  description="Researches carbon capture methods.",  tools=[google\_search],  # Store result in state for the merger agent  output\_key="carbon\_capture\_result"  )  # --- 2. Create the ParallelAgent (Runs researchers concurrently) ---  # This agent orchestrates the concurrent execution of the researchers.  # It finishes once all researchers have completed and stored their results in state.  parallel\_research\_agent = ParallelAgent(  name="ParallelWebResearchAgent",  sub\_agents=[researcher\_agent\_1, researcher\_agent\_2, researcher\_agent\_3],  description="Runs multiple research agents in parallel to gather information."  )  # --- 3. Define the Merger Agent (Runs \*after\* the parallel agents) ---  # This agent takes the results stored in the session state by the parallel agents  # and synthesizes them into a single, structured response with attributions.  merger\_agent = LlmAgent(  name="SynthesisAgent",  model=GEMINI\_MODEL, # Or potentially a more powerful model if needed for synthesis  instruction="""You are an AI Assistant responsible for combining research findings into a structured report.  Your primary task is to synthesize the following research summaries, clearly attributing findings to their source areas. Structure your response using headings for each topic. Ensure the report is coherent and integrates the key points smoothly.  \*\*Crucially: Your entire response MUST be grounded \*exclusively\* on the information provided in the 'Input Summaries' below. Do NOT add any external knowledge, facts, or details not present in these specific summaries.\*\*  \*\*Input Summaries:\*\*  \* \*\*Renewable Energy:\*\*  {renewable\_energy\_result}  \* \*\*Electric Vehicles:\*\*  {ev\_technology\_result}  \* \*\*Carbon Capture:\*\*  {carbon\_capture\_result}  \*\*Output Format:\*\*  ## Summary of Recent Sustainable Technology Advancements  ### Renewable Energy Findings  (Based on RenewableEnergyResearcher's findings)  [Synthesize and elaborate \*only\* on the renewable energy input summary provided above.]  ### Electric Vehicle Findings  (Based on EVResearcher's findings)  [Synthesize and elaborate \*only\* on the EV input summary provided above.]  ### Carbon Capture Findings  (Based on CarbonCaptureResearcher's findings)  [Synthesize and elaborate \*only\* on the carbon capture input summary provided above.]  ### Overall Conclusion  [Provide a brief (1-2 sentence) concluding statement that connects \*only\* the findings presented above.]  Output \*only\* the structured report following this format. Do not include introductory or concluding phrases outside this structure, and strictly adhere to using only the provided input summary content.  """,  description="Combines research findings from parallel agents into a structured, cited report, strictly grounded on provided inputs.",  # No tools needed for merging  # No output\_key needed here, as its direct response is the final output of the sequence  )  # --- 4. Create the SequentialAgent (Orchestrates the overall flow) ---  # This is the main agent that will be run. It first executes the ParallelAgent  # to populate the state, and then executes the MergerAgent to produce the final output.  sequential\_pipeline\_agent = SequentialAgent(  name="ResearchAndSynthesisPipeline",  # Run parallel research first, then merge  sub\_agents=[parallel\_research\_agent, merger\_agent],  description="Coordinates parallel research and synthesizes the results."  )  root\_agent = sequential\_pipeline\_agent |

This code defines a multi-agent system used to research and synthesize information on sustainable technology advancements. It sets up three LlmAgent instances to act as specialized researchers. ResearcherAgent\_1 focuses on renewable energy sources, ResearcherAgent\_2 researches electric vehicle technology, and ResearcherAgent\_3 investigates carbon capture methods. Each researcher agent is configured to use a GEMINI\_MODEL and the google\_search tool. They are instructed to summarize their findings concisely (1-2 sentences) and store these summaries in the session state using output\_key.

A ParallelAgent named ParallelWebResearchAgent is then created to run these three researcher agents concurrently. This allows the research to be conducted in parallel, potentially saving time. The ParallelAgent completes its execution once all its sub-agents (the researchers) have finished and populated the state.

Next, a MergerAgent (also an LlmAgent) is defined to synthesize the research results. This agent takes the summaries stored in the session state by the parallel researchers as input. Its instruction emphasizes that the output must be strictly based only on the provided input summaries, prohibiting the addition of external knowledge. The MergerAgent is designed to structure the combined findings into a report with headings for each topic and a brief overall conclusion.

Finally, a SequentialAgent named ResearchAndSynthesisPipeline is created to orchestrate the entire workflow. As the primary controller, this main agent first executes the ParallelAgent to perform the research. Once the ParallelAgent is complete, the SequentialAgent then executes the MergerAgent to synthesize the collected information. The sequential\_pipeline\_agent is set as the root\_agent, representing the entry point for running this multi-agent system. The overall process is designed to efficiently gather information from multiple sources in parallel and then combine it into a single, structured report.

**At a Glance**

**What:** Many agentic workflows involve multiple sub-tasks that must be completed to achieve a final goal. A purely sequential execution, where each task waits for the previous one to finish, is often inefficient and slow. This latency becomes a significant bottleneck when tasks depend on external I/O operations, such as calling different APIs or querying multiple databases. Without a mechanism for concurrent execution, the total processing time is the sum of all individual task durations, hindering the system's overall performance and responsiveness.

**Why:** The Parallelization pattern provides a standardized solution by enabling the simultaneous execution of independent tasks. It works by identifying components of a workflow, like tool usages or LLM calls, that do not rely on each other's immediate outputs. Agentic frameworks like LangChain and the Google ADK provide built-in constructs to define and manage these concurrent operations. For instance, a main process can invoke several sub-tasks that run in parallel and wait for all of them to complete before proceeding to the next step. By running these independent tasks at the same time rather than one after another, this pattern drastically reduces the total execution time.

**Rule of thumb:** Use this pattern when a workflow contains multiple independent operations that can run simultaneously, such as fetching data from several APIs, processing different chunks of data, or generating multiple pieces of content for later synthesis.

**Visual summary**

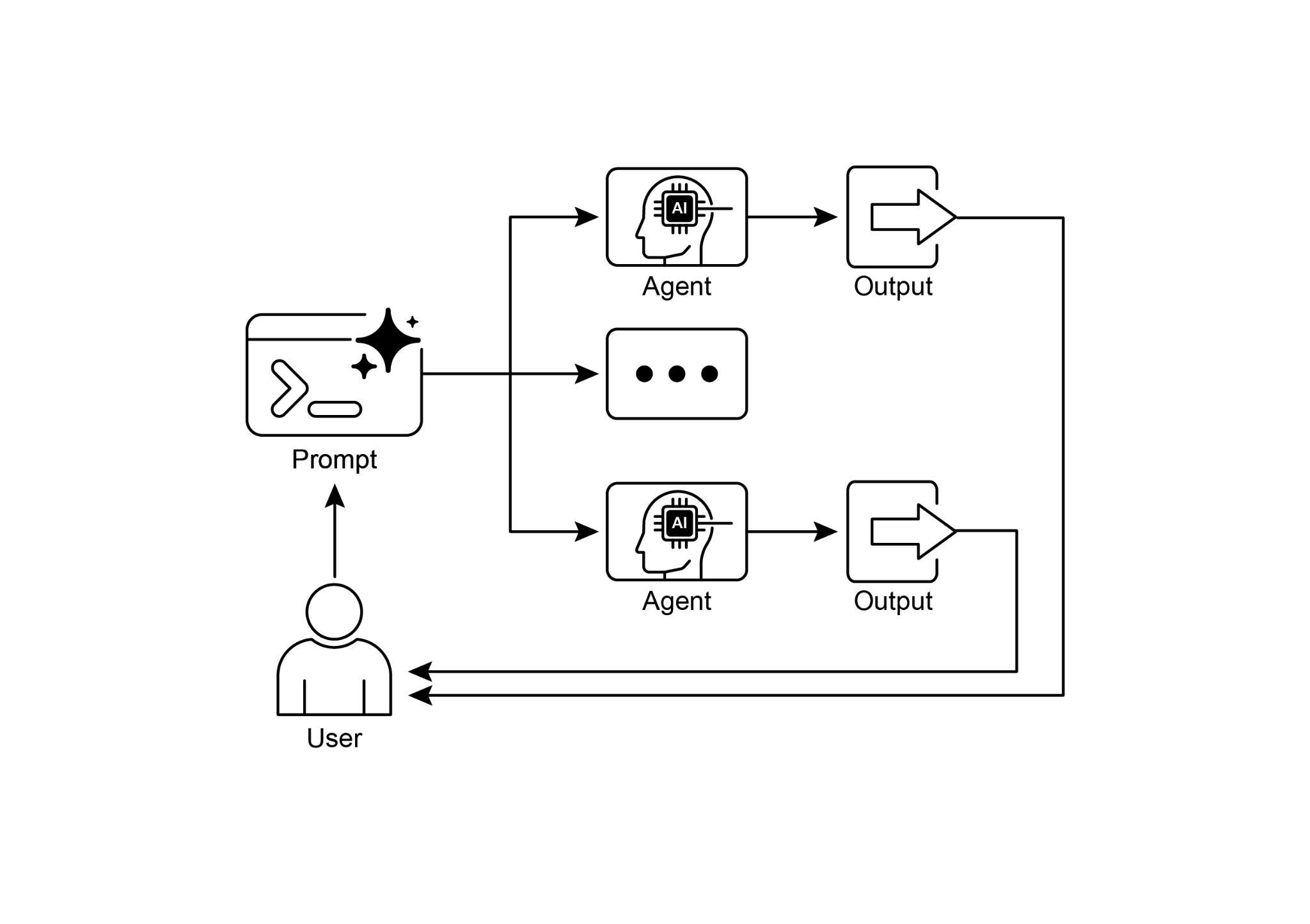


Fig.2: Parallelization design pattern

**Key Takeaways**

Here are the key takeaways:

* Parallelization is a pattern for executing independent tasks concurrently to improve efficiency.
* It is particularly useful when tasks involve waiting for external resources, such as API calls.
* The adoption of a concurrent or parallel architecture introduces substantial complexity and cost, impacting key development phases such as design, debugging, and system logging.
* Frameworks like LangChain and Google ADK provide built-in support for defining and managing parallel execution.
* In LangChain Expression Language (LCEL), RunnableParallel is a key construct for running multiple runnables side-by-side.
* Google ADK can facilitate parallel execution through LLM-Driven Delegation, where a Coordinator agent's LLM identifies independent sub-tasks and triggers their concurrent handling by specialized sub-agents.
* Parallelization helps reduce overall latency and makes agentic systems more responsive for complex tasks.

**Conclusion**

The parallelization pattern is a method for optimizing computational workflows by concurrently executing independent sub-tasks. This approach reduces overall latency, particularly in complex operations that involve multiple model inferences or calls to external services.

Frameworks provide distinct mechanisms for implementing this pattern. In LangChain, constructs like RunnableParallel are used to explicitly define and execute multiple processing chains simultaneously. In contrast, frameworks like the Google Agent Developer Kit (ADK) can achieve parallelization through multi-agent delegation, where a primary coordinator model assigns different sub-tasks to specialized agents that can operate concurrently.

By integrating parallel processing with sequential (chaining) and conditional (routing) control flows, it becomes possible to construct sophisticated, high-performance computational systems capable of efficiently managing diverse and complex tasks.

**References**

Here are some resources for further reading on the Parallelization pattern and related concepts:

1. LangChain Expression Language (LCEL) Documentation (Parallelism): <https://python.langchain.com/docs/concepts/lcel/>
2. Google Agent Developer Kit (ADK) Documentation (Multi-Agent Systems): <https://google.github.io/adk-docs/agents/multi-agents/>
3. Python asyncio Documentation: <https://docs.python.org/3/library/asyncio.html>

**第三章\_并行化**

第3章：并行化

**并行化模式概述**

在前面的章节中，我们探讨了用于顺序工作流的提示链和用于动态决策以及不同路径之间转换的路由。虽然这些模式至关重要，但许多复杂的智能体任务涉及多个子任务，这些子任务可以*同时*执行，而不是依次执行。这就是**并行化**模式变得至关重要的地方。

并行化是指同时执行多个组件，如大语言模型调用、工具使用，甚至整个子智能体（见图1）。并行执行不是等待一个步骤完成后再开始下一个步骤，而是允许独立任务同时运行，从而显著减少可分解为独立部分的任务的整体执行时间。

假设有一个智能体，其设计目的是研究某个主题并总结研究结果。顺序式方法可能会：

1. 搜索源A。
2. 总结资料A。
3. 搜索源B。
4. 总结资料B。
5. 从摘要A和摘要B中综合出最终答案。

相反，并行方法可以：

1. 搜索源A*并*同时搜索源B。
2. 两个搜索完成后，同时总结来源A*和*来源B。
3. 从摘要A和摘要B中综合出最终答案（此步骤通常是按顺序进行的，需等待并行步骤完成）。

核心思想是识别工作流中不依赖于其他部分输出的部分，并并行执行它们。在处理有延迟的外部服务（如 API 或数据库）时，这特别有效，因为你可以同时发出多个请求。

实现并行化通常需要支持异步执行或多线程/多进程的框架。现代的智能体框架在设计时就考虑到了异步操作，使你能够轻松定义可以并行运行的步骤。

图1. 使用子智能体进行并行化的示例

像LangChain、LangGraph和Google ADK这样的框架提供了并行执行的机制。在LangChain表达式语言（LCEL）中，你可以通过使用|（用于顺序执行）等运算符组合可运行对象，并通过构建你的链或图使其具有并发执行的分支来实现并行执行。LangGraph凭借其图结构，允许你定义多个节点，这些节点可以从单个状态转换中执行，从而有效地在工作流中实现并行分支。Google ADK提供了强大的原生机制，以促进和管理代理的并行执行，显著提高了复杂多代理系统的效率和可扩展性。ADK框架的这种内在能力使开发人员能够设计和实现多个代理可以并发而非顺序运行的解决方案。

并行化模式对于提高智能体系统的效率和响应能力至关重要，尤其是在处理涉及多个独立查找、计算或与外部服务交互的任务时。它是优化复杂智能体工作流性能的关键技术。

**实际应用与用例**

并行化是一种强大的模式，可用于优化各种应用程序中的代理性能：

1. 信息收集与研究：

同时从多个来源收集信息是一个典型的用例。

* **用例：**一名代理人员正在对一家公司进行调研。
* **并行任务：**同时搜索新闻文章、提取股票数据、查看社交媒体提及情况并查询公司数据库。
* **优点：**比顺序查找更快地收集全面视图。

2. 数据处理与分析：

同时应用不同的分析技术或处理不同的数据段。

* **用例：**分析客户反馈的代理。
* **并行任务：**在一批反馈条目中同时运行情感分析、提取关键词、对反馈进行分类并识别紧急问题。
* **优点：**快速提供多方面分析。

3. 多API或工具交互：

调用多个独立的 API 或工具来收集不同类型的信息或执行不同的操作。

* **用例：**旅行规划代理。
* **并行任务：**同时查询航班价格、搜索酒店可用性、查找当地活动并寻找餐厅推荐。
* **优点：**更快地呈现完整的旅行计划。

4. 多组件内容生成：

并行生成复杂内容的不同部分。

* **用例：**代理创建营销电子邮件。
* **并行任务：**同时生成主题行、起草邮件正文、查找相关图片并创建行动呼吁按钮文本。
* **优点：**更高效地组装最终电子邮件。

5. 验证与确认：

同时执行多个独立的检查或验证。

* **用例：**代理验证用户输入。
* **并行任务：**同时检查电子邮件格式、验证电话号码、对照数据库核实地址并检查是否存在亵渎性内容。
* **优点：**对输入有效性提供更快的反馈。

6. 多模态处理：

同时处理同一输入的不同模态（文本、图像、音频）。

* **用例：**一个智能体分析包含文本和图像的社交媒体帖子。
* **并行任务：**分析文本的情感和关键词*以及*同时分析图像中的物体和场景描述。
* **优点：**能更快地整合来自不同模态的见解。

7. A/B测试或多选项生成：

并行生成响应或输出的多个变体，以选择最佳的一个。

* **用例：**一个智能体生成不同的创意文本选项。
* **并行任务：**使用略有不同的提示或模型同时为一篇文章生成三个不同的标题。
* **优点：**便于快速比较和选择最佳方案。

并行化是智能体设计中的一项基本优化技术，它允许开发者通过利用独立任务的并发执行来构建性能更高、响应更灵敏的应用程序。

**实践代码示例（LangChain）**

LangChain框架内的并行执行由LangChain表达式语言（LCEL）实现。主要方法是在字典或列表结构中组织多个可运行组件。当这个集合作为输入传递给链中的后续组件时，LCEL运行时会并发执行其中包含的可运行组件。

在LangGraph的语境中，这一原则应用于图的拓扑结构。通过对图进行架构设计，使得多个没有直接顺序依赖关系的节点可以从单个公共节点启动，从而定义并行工作流。这些并行路径在图中的后续收敛点聚合结果之前独立执行。

以下实现展示了一个使用LangChain框架构建的并行处理工作流。该工作流旨在响应单个用户查询，同时执行两个独立操作。这些并行进程被实例化为不同的链或函数，其各自的输出随后被聚合为一个统一的结果。

此实现的先决条件包括安装必要的Python包，如langchain、langchain-community，以及像langchain-openai这样的模型提供程序库。此外，必须在本地环境中配置所选语言模型的有效API密钥以进行身份验证。

|  |
| --- |
| import os  导入 asyncio  从 typing 导入 Optional  from langchain\_openai import ChatOpenAI  from langchain\_core.prompts import ChatPromptTemplate  从 langchain\_core.output\_parsers 导入 StrOutputParser  从 langchain\_core.runnables 导入 Runnable、RunnableParallel、RunnablePassthrough  # --- 配置 ---  # 确保已设置 API 密钥环境变量（例如，OPENAI\_API\_KEY）  try:  llm: Optional[ChatOpenAI] = ChatOpenAI(model="gpt-4o-mini", temperature=0.7)    except Exception as e:  print(f"初始化语言模型时出错: {e}")  llm = None  # --- 定义独立链 ---  #这三条链代表了可以并行执行的不同任务。  summarize\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("系统", "简洁地总结以下主题："),  ("用户", "{主题}")  ])  | 大语言模型  | StrOutputParser()  )  questions\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("系统", "针对以下主题生成三个有趣的问题："),  ("用户", "{主题}")  ])  | 大语言模型  | StrOutputParser()  )  terms\_chain: Runnable = (  ChatPromptTemplate.from\_messages([  ("系统", "从以下主题中识别5-10个关键词，用逗号分隔："),  ("用户", "{主题}")  ])  | 大语言模型  | StrOutputParser()  )  # --- 构建并行 + 合成链 ---  # 1. 定义要并行运行的任务块。这些任务的结果，  #连同原始主题，将被输入到下一步。  map\_chain = RunnableParallel(  {  "summary": summarize\_chain,  "问题": questions\_chain,  "关键术语": terms\_chain,  "topic": RunnablePassthrough(), # 直接传递原始主题  }  )  # 2. 定义最终合成提示，该提示将合并并行结果。  synthesis\_prompt = ChatPromptTemplate.from\_messages([  ("系统", """根据以下信息：  摘要：{summary}  相关问题：{questions}  关键术语：{key\_terms}  综合生成一个全面的答案。  ("用户", "原始主题: {topic}")  ])  # 3. 通过直接连接并行结果来构建完整的链  # 进入合成提示，随后是大语言模型（LLM）和输出解析器。  full\_parallel\_chain = map\_chain | synthesis\_prompt | llm | StrOutputParser()  # --- 运行链 ---  async def run\_parallel\_example(topic: str) -> None:  """  异步调用具有特定主题的并行处理链  并打印合成结果。  参数：  主题：由LangChain链处理的输入主题。  """  if not llm:  print("大语言模型未初始化。无法运行示例。")  返回  print(f"\n--- 正在运行主题为 '{topic}' 的并行 LangChain 示例 ---")  try:  # `ainvoke`的输入是单个 'topic' 字符串，  # 然后传递给 `map\_chain` 中的每个可运行对象。  response = await full\_parallel\_chain.ainvoke(topic)  print("\n---最终响应---")  print(response)  except Exception as e:  print(f"\n链执行期间发生错误: {e}")  if \_\_name\_\_ == "\_\_main\_\_":  test\_topic = "太空探索的历史"  # 在 Python 3.7 及更高版本中，asyncio.run 是运行异步函数的标准方法。  asyncio.run(run\_parallel\_example(test\_topic)) |

所提供的Python代码实现了一个LangChain应用程序，该程序旨在通过利用并行执行来高效处理给定主题。请注意，asyncio提供的是并发，而非并行性。它通过使用事件循环在单线程上实现这一点，当一个任务空闲（例如，等待网络请求）时，事件循环会智能地在任务之间切换。这产生了多个任务同时推进的效果，但代码本身仍然仅由一个线程执行，受Python全局解释器锁（GIL）的限制。

代码首先从langchain\_openai和langchain\_core导入必要的模块，包括语言模型、提示、输出解析和可运行结构的组件。代码尝试初始化一个ChatOpenAI实例，具体使用“gpt-4o-mini”模型，并指定温度以控制创造力。在语言模型初始化期间，使用try-except块来确保健壮性。然后定义了三个独立的LangChain“链”，每个链都旨在对输入主题执行不同的任务。第一个链用于简洁地总结主题，使用系统消息和包含主题占位符的用户消息。第二个链配置为生成与主题相关的三个有趣问题。第三个链设置为从输入主题中识别5到10个关键词，并要求用逗号分隔。这些独立的链中的每一个都由一个针对其特定任务定制的ChatPromptTemplate、初始化的语言模型和一个将输出格式化为字符串的StrOutputParser组成。

然后构建一个 RunnableParallel 块来捆绑这三个链，使它们能够同时执行。这个并行可运行对象还包含一个 RunnablePassthrough，以确保原始输入主题可用于后续步骤。为最终的综合步骤定义了一个单独的 ChatPromptTemplate，它将摘要、问题、关键词和原始主题作为输入，以生成全面的答案。名为 full\_parallel\_chain 的完整端到端处理链是通过将 map\_chain（并行块）按顺序连接到综合提示，然后连接语言模型和输出解析器来创建的。提供了一个异步函数 run\_parallel\_example 来演示如何调用这个 full\_parallel\_chain。这个函数将主题作为输入，并使用 invoke 来运行异步链。最后，标准的 Python if \_\_name\_\_ == "\_\_main\_\_": 块展示了如何使用 asyncio.run 来管理异步执行，以一个示例主题（在本例中为 "太空探索的历史"）执行 run\_parallel\_example。

本质上，这段代码建立了一个工作流，在这个工作流中，针对给定主题同时进行多个大语言模型调用（用于总结、提问和术语处理），然后通过最后一次大语言模型调用将它们的结果合并。这展示了使用LangChain在智能体工作流中并行化的核心思想。

**实践代码示例（谷歌ADK）**

好了，现在让我们把注意力转向一个具体的例子，该例子将说明这些概念在谷歌ADK框架中的应用。我们将研究如何应用ADK原语（如ParallelAgent和SequentialAgent）来构建一个利用并发执行以提高效率的代理流程。

|  |
| --- |
| from google.adk.agents import LlmAgent, ParallelAgent, SequentialAgent  from google.adk.tools import google\_search  GEMINI\_MODEL="gemini-2.0-flash"  # --- 1. 定义研究人员子智能体（并行运行） ---  # 研究员1：可再生能源  researcher\_agent\_1 = LlmAgent(  name="可再生能源研究员",  model=GEMINI\_MODEL,  instruction="你是一位专门从事能源领域的AI研究助理。  研究“可再生能源”的最新进展。  使用提供的谷歌搜索工具。  简洁地总结你的主要发现（1 - 2句话）。  仅输出摘要。  """,  description="研究可再生能源。",  tools=[google\_search],  # 将结果存储在合并代理的状态中  output\_key="可再生能源结果"  )  # 研究员2：电动汽车  researcher\_agent\_2 = LlmAgent(  name="EV研究员",  model=GEMINI\_MODEL,  instruction="你是一位专门研究交通领域的AI研究助理。  研究“电动汽车技术”的最新发展。  使用提供的谷歌搜索工具。  简洁地总结你的主要发现（1 - 2句话）。  仅输出摘要。  """,  description="研究电动汽车技术。",  tools=[google\_search],  # 将结果存储在合并代理的状态中  output\_key="ev\_technology\_result"  )  #研究员3：碳捕获  researcher\_agent\_3 = LlmAgent(  name="碳捕获研究员",  model=GEMINI\_MODEL,  instruction="你是一位专门研究气候解决方案的AI研究助理。  研究“碳捕获方法”的现状。  使用提供的谷歌搜索工具。  简洁地总结你的主要发现（1 - 2句话）。  仅输出摘要。  """,  description="研究碳捕获方法。",  tools=[google\_search],  # 将结果存储在合并代理的状态中  output\_key="carbon\_capture\_result"  )  # --- 2. 创建并行代理（同时运行研究人员） ---  # 此代理协调研究人员的并发执行。  # 一旦所有研究人员完成并将其结果存储在状态中，它就会结束。  parallel\_research\_agent = ParallelAgent(  name="并行网络研究代理",  sub\_agents=[researcher\_agent\_1, researcher\_agent\_2, researcher\_agent\_3],  description="并行运行多个研究代理以收集信息。"  )  # --- 3. 定义合并代理（在并行代理 \*之后\* 运行） ---  # 此代理获取并行代理存储在会话状态中的结果  #并将它们综合成一个有出处的单一结构化回复。  merger\_agent = LlmAgent(  name="SynthesisAgent",  model=GEMINI\_MODEL, # 或者如果合成需要，可能使用更强大的模型  instruction="你是一个负责将研究结果整合为结构化报告的AI助手。  你的主要任务是综合以下研究摘要，明确将研究结果归因于其来源领域。使用每个主题的标题来组织你的回应。确保报告连贯一致，并能顺利整合关键要点。  至关重要的是：你的整个回复必须\*完全\*基于以下“输入摘要”中提供的信息。不要添加这些特定摘要中未包含的任何外部知识、事实或细节。  \*\*输入摘要：\*\*  \* \*\*可再生能源：\*\*  {可再生能源结果}  \* \*\*电动汽车：\*\*  {ev\_technology\_result}  \* \*\*碳捕获\*\*：  {碳捕获结果}  \*\*输出格式：\*\*  ## 近期可持续技术进展综述  ###可再生能源研究结果  （基于可再生能源研究员的发现）  [仅对上述提供的可再生能源输入摘要进行综合和详细阐述。]  ### 电动汽车研究结果  (基于EVResearcher的研究结果)  [仅对上述提供的 EV 输入摘要进行综合和阐述。]  ### 碳捕获研究结果  (基于CarbonCaptureResearcher的研究结果)  [仅对上述提供的碳捕获输入摘要进行综合和详细阐述。]  ### 总体结论  [提供一个简短（1-2句）的结论性陈述，仅与上述呈现的研究结果相关联。]  仅按照以下格式输出结构化报告。不要在该结构之外包含引言或结论性短语，并且严格仅使用提供的输入摘要内容。  """,  description="将来自并行代理的研究结果整合到一份结构严谨、有引用依据的报告中，严格基于所提供的输入。",  # 合并无需工具  # 此处不需要输出键，因为其直接响应是序列的最终输出  )  # --- 4. 创建顺序代理（编排整体流程） ---  # 这是将被运行的主代理。它首先执行并行代理  # 填充状态，然后执行合并代理以生成最终输出。  sequential\_pipeline\_agent = SequentialAgent(  name="研究与合成管道",  # 先并行开展研究，然后合并  sub\_agents=[parallel\_research\_agent, merger\_agent],  description="协调平行研究并综合研究结果。"  )  root\_agent = sequential\_pipeline\_agent |

此代码定义了一个多智能体系统，用于研究和综合可持续技术进步方面的信息。它设置了三个大语言模型智能体实例，作为专业研究人员。研究智能体1专注于可再生能源，研究智能体2研究电动汽车技术，研究智能体3调查碳捕获方法。每个研究智能体都配置为使用双子座模型和谷歌搜索工具。它们被要求简洁地总结其研究结果（1 - 2句话），并使用输出键将这些总结存储在会话状态中。

随后创建了一个名为ParallelWebResearchAgent的并行代理，以同时运行这三个研究人员代理。这使得研究能够并行进行，有可能节省时间。一旦其所有子代理（研究人员）完成任务并填充了状态，并行代理就完成了执行。

接下来，定义了一个合并代理（也是一个大语言模型代理）来综合研究结果。该代理将并行研究人员存储在会话状态中的摘要作为输入。其指令强调，输出必须严格仅基于所提供的输入摘要，禁止添加外部知识。合并代理旨在将合并后的结果组织成一份报告，为每个主题设置标题，并给出简要的总体结论。

最后，创建了一个名为ResearchAndSynthesisPipeline的顺序代理，以编排整个工作流程。作为主要控制器，这个主代理首先执行并行代理来进行研究。一旦并行代理完成，顺序代理就会执行合并代理来综合收集到的信息。顺序管道代理被设置为根代理，代表运行这个多智能体系统的切入点。整个过程旨在高效地从多个来源并行收集信息，然后将其整合为一份单一的、结构化的报告。

**概览**

**问题：**许多智能工作流涉及多个子任务，必须完成这些子任务才能实现最终目标。纯顺序执行（即每个任务都要等待前一个任务完成）往往效率低下且速度缓慢。当任务依赖于外部I/O操作（如调用不同的API或查询多个数据库）时，这种延迟就会成为一个重大瓶颈。如果没有并发执行机制，总流转时长就是所有单个任务时长的总和，这会阻碍系统的整体性能和响应能力。

**原因：**并行化模式通过支持独立任务的同时执行，提供了一种标准化的解决方案。它的工作原理是识别工作流中彼此不依赖即时输出的组件，如工具使用或大语言模型调用。像LangChain和谷歌ADK这样的智能体框架提供了内置结构来定义和管理这些并发操作。例如，一个主进程可以调用多个并行运行的子任务，并在继续下一步之前等待所有子任务完成。通过同时运行这些独立任务而不是依次运行，这种模式大大减少了总执行时间。

**经验法则：**当工作流包含多个可以同时运行的独立操作时，使用此模式，例如从多个 API 获取数据、处理不同的数据块，或生成多个内容片段以供后续合成。

**可视化总结**

图2：并行化设计模式

**要点总结**

以下是关键收获：

* 并行化是一种通过同时执行独立任务来提高效率的模式。
* 当任务涉及等待外部资源（如 API 调用）时，它特别有用。
* 采用并发或并行架构会带来极大的复杂性和成本，影响设计、调试和系统日志记录等关键开发阶段。
* 像LangChain和Google ADK这样的框架提供了对定义和管理并行执行的内置支持。
* 在LangChain表达式语言（LCEL）中，RunnableParallel是一个关键的构造，用于并行运行多个可运行对象。
* 谷歌ADK可以通过大语言模型驱动的委托来促进并行执行，其中协调器代理的大语言模型识别独立的子任务，并触发专业子代理对其进行并发处理。
* 并行化有助于减少整体延迟，并使自主系统在处理复杂任务时更具响应性。

**结论**

并行化模式是一种通过并发执行独立子任务来优化计算工作流的方法。这种方法可以减少整体延迟，特别是在涉及多个模型推理或调用外部服务的复杂操作中。

不同的框架为实现这一模式提供了独特的机制。在LangChain中，像RunnableParallel这样的构造被用来显式地定义和同时执行多个处理链。相比之下，像谷歌代理开发工具包（ADK）这样的框架可以通过多代理委托实现并行化，其中一个主要的协调器模型将不同的子任务分配给可以同时运行的专业代理。

通过将并行处理与顺序（链式）和条件（路由）控制流相结合，就有可能构建出能够高效管理多样且复杂任务的复杂高性能计算系统。

**参考文献**

以下是一些关于并行化模式及相关概念的进一步阅读资源：

1. LangChain表达式语言（LCEL）文档（并行性）：<https://python.langchain.com/docs/concepts/lcel/>
2. Google Agent Developer Kit (ADK) 文档 (多智能体系统): <https://google.github.io/adk-docs/agents/multi-agents/>
3. Python 异步I/O文档：<https://docs.python.org/3/library/asyncio.html>