```
from concurrent.futures import ThreadPoolExecutor, as_completed
from typing import Any, Callable, List, Optional

from swarms import Agent
from swarms.structs.base_swarm import BaseSwarm
from swarms.utils.loguru_logger import logger


class MonteCarloSwarm(BaseSwarm):
    """
    MonteCarloSwarm leverages multiple agents to collaborate in a Monte Carlo fashion.
    Each agent's output is passed to the next, refining the result progressively.
    Supports parallel execution, dynamic agent selection, and custom result aggregation.

    Attributes:
        agents (List[Agent]): A list of agents that will participate in the swarm.
        parallel (bool): If True, agents will run in parallel.
        result_aggregator (Callable[[List[Any]], Any]): A function to aggregate results from agents.
        max_workers (Optional[int]): The maximum number of threads for parallel execution.
    """

    def __init__(
        self,
        agents: List[Agent],
        parallel: bool = False,
        result_aggregator: Optional[
```

```
        Callable[[List[Any]], Any]
    ] = None,
    max_workers: Optional[int] = None,
    *args,
    **kwargs,
) -> None:
    """
    Initializes the MonteCarloSwarm with a list of agents.

    Args:
        agents (List[Agent]): A list of agents to include in the swarm.
        parallel (bool): If True, agents will run in parallel. Default is False.
        result_aggregator (Optional[Callable[[List[Any]], Any]]): A function to aggregate results from agents.
        max_workers (Optional[int]): The maximum number of threads for parallel execution.
    """
    super().__init__(agents=agents, *args, **kwargs)

    if not agents:
        raise ValueError("The agents list cannot be empty.")

    self.agents = agents
    self.parallel = parallel
    self.result_aggregator = (
        result_aggregator or self.default_aggregator
    )
```

```
    self.max_workers = max_workers or len(agents)

def run(self, task: str) -> Any:
    """
    Runs the MonteCarloSwarm with the given input, passing the output of each agent
    to the next one in the list or running agents in parallel.

    Args:
        task (str): The initial input to provide to the first agent.

    Returns:
        Any: The final output after all agents have processed the input.
    """
    logger.info(
        f"Starting MonteCarloSwarm with parallel={self.parallel}"
    )

    if self.parallel:
        results = self._run_parallel(task)
    else:
        results = self._run_sequential(task)

    final_output = self.result_aggregator(results)
    logger.info(
        f"MonteCarloSwarm completed. Final output: {final_output}"
    )
```

```
    return final_output

def _run_sequential(self, task: str) -> List[Any]:
    """
    Runs the agents sequentially, passing each agent's output to the next.

    Args:
        task (str): The initial input to provide to the first agent.

    Returns:
        List[Any]: A list of results from each agent.
    """
    results = []
    current_input = task
    for i, agent in enumerate(self.agents):
        logger.info(f"Agent {i + 1} processing sequentially...")
        current_output = agent.run(current_input)
        results.append(current_output)
        current_input = current_output
    return results

def _run_parallel(self, task: str) -> List[Any]:
    """
    Runs the agents in parallel, each receiving the same initial input.

    Args:
```

```
        task (str): The initial input to provide to all agents.

    Returns:
        List[Any]: A list of results from each agent.
    """
    results = []
    with ThreadPoolExecutor(
        max_workers=self.max_workers
    ) as executor:
        future_to_agent = {
            executor.submit(agent.run, task): agent
            for agent in self.agents
        }
        for future in as_completed(future_to_agent):
            try:
                result = future.result()
                results.append(result)
                logger.info(
                    f"Agent completed with result: {result}"
                )
            except Exception as e:
                logger.error(f"Agent encountered an error: {e}")
                results.append(None)
    return results

@staticmethod
```

```
    def default_aggregator(results: List[Any]) -> Any:
        """
        Default result aggregator that returns the last result.

        Args:
            results (List[Any]): A list of results from agents.

        Returns:
            Any: The final aggregated result.
        """
        return results


def average_aggregator(results: List[float]) -> float:
    return sum(results) / len(results) if results else 0.0


# # Example usage
# if __name__ == "__main__":
#     # Get the OpenAI API key from the environment variable
#     api_key = os.getenv("OPENAI_API_KEY")

#     # Create an instance of the OpenAIChat class
#     model = OpenAIChat(
#         api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
#     )
```

```
# # Initialize the agents
# agents_list = [
#     Agent(
#         agent_name="Financial-Analysis-Agent-1",
#         system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=False,
#         dashboard=False,
#         verbose=True,
#         streaming_on=True,
#         dynamic_temperature_enabled=True,
#         saved_state_path="finance_agent_1.json",
#         retry_attempts=3,
#         context_length=200000,
#     ),
#     Agent(
#         agent_name="Financial-Analysis-Agent-2",
#         system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=False,
#         dashboard=False,
#         verbose=True,
#         streaming_on=True,
```

```
#             dynamic_temperature_enabled=True,
#             saved_state_path="finance_agent_2.json",
#             retry_attempts=3,
#             context_length=200000,
#         ),
#         # Add more agents as needed
#     ]

#     # Initialize the MonteCarloSwarm with parallel execution enabled
#     swarm = MonteCarloSwarm(
#         agents=agents_list, parallel=True, max_workers=2
#     )

#     # Run the swarm with an initial query
#     final_output = swarm.run(
#         "What are the components of a startup's stock incentive equity plan?"
#     )
#     print("Final output:", final_output)



# import os
# from swarms import Agent

# from typing import List, Union, Callable
# from collections import Counter
```

```
# # Aggregation functions



# def aggregate_most_common_result(results: List[str]) -> str:
#     """
#     Aggregate results using the most common result.

#     Args:
#         results (List[str]): List of results from each iteration.

#     Returns:
#         str: The most common result.
#     """
#     result_counter = Counter(results)
#     most_common_result = result_counter.most_common(1)[0][0]
#     return most_common_result



# def aggregate_weighted_vote(results: List[str], weights: List[int]) -> str:
#     """
#     Aggregate results using a weighted voting system.

#     Args:
#         results (List[str]): List of results from each iteration.
#         weights (List[int]): List of weights corresponding to each result.
```

```
#     Returns:
#         str: The result with the highest weighted vote.
#     """
#     weighted_results = Counter()
#     for result, weight in zip(results, weights):
#         weighted_results[result] += weight

#     weighted_result = weighted_results.most_common(1)[0][0]
#     return weighted_result



# def aggregate_average_numerical(results: List[Union[str, float]]) -> float:
#     """
#     Aggregate results by averaging numerical outputs.

#     Args:
#         results (List[Union[str, float]]): List of numerical results from each iteration.

#     Returns:
#         float: The average of the numerical results.
#     """
#     numerical_results = [
#         float(result) for result in results if is_numerical(result)
#     ]
#     if numerical_results:
#         return sum(numerical_results) / len(numerical_results)
```

```
#     else:
#         return float("nan")  # or handle non-numerical case as needed



# def aggregate_consensus(results: List[str]) -> Union[str, None]:
#     """
#     Aggregate results by checking if there's a consensus (all results are the same).

#     Args:
#         results (List[str]): List of results from each iteration.

#     Returns:
#         Union[str, None]: The consensus result if there is one, otherwise None.
#     """
#     if all(result == results[0] for result in results):
#         return results[0]
#     else:
#         return None  # or handle lack of consensus as needed



# def is_numerical(value: str) -> bool:
#     """
#     Check if a string can be interpreted as a numerical value.

#     Args:
#         value (str): The string to check.
```

```
#     Returns:
#         bool: True if the string is numerical, otherwise False.
#     """
#     try:
#         float(value)
#         return True
#     except ValueError:
#         return False



# # MonteCarloSwarm class



# class MonteCarloSwarm:
#     def __init__(
#         self,
#         agents: List[Agent],
#         iterations: int = 100,
#         aggregator: Callable = aggregate_most_common_result,
#     ):
#         self.agents = agents
#         self.iterations = iterations
#         self.aggregator = aggregator

#     def run(self, task: str) -> Union[str, float, None]:
```

```
#         """
#         Execute the Monte Carlo swarm, passing the output of each agent to the next.
#         The final result is aggregated over multiple iterations using the provided aggregator.

#         Args:
#             task (str): The task for the swarm to execute.

#         Returns:
#             Union[str, float, None]: The final aggregated result.
#         """
#         aggregated_results = []

#         for i in range(self.iterations):
#             result = task
#             for agent in self.agents:
#                 result = agent.run(result)
#             aggregated_results.append(result)

#         # Apply the selected aggregation function
#         final_result = self.aggregator(aggregated_results)
#         return final_result


# # Example usage:

# # Assuming you have the OpenAI API key set up and agents defined
```

```
# api_key = os.getenv("OPENAI_API_KEY")
# model = OpenAIChat(
#     api_key=api_key, model_name="gpt-4o-mini", temperature=0.1
# )

# agent1 = Agent(
#     agent_name="Agent1",
#     system_prompt="System prompt for agent 1",
#     llm=model,
#     max_loops=1,
#     verbose=True,
# )

# agent2 = Agent(
#     agent_name="Agent2",
#     system_prompt="System prompt for agent 2",
#     llm=model,
#     max_loops=1,
#     verbose=True,
# )

# # Create a MonteCarloSwarm with the agents and a selected aggregation function
# swarm = MonteCarloSwarm(
#     agents=[agent1, agent2],
#     iterations=1,
#     aggregator=aggregate_weighted_vote,
```

```
# )


# # Run the swarm on a specific task
# final_output = swarm.run(
#     "What are the components of a startup's stock incentive plan?"
# )
# print("Final Output:", final_output)


# # You can easily switch the aggregation function by passing a different one to the constructor:
# # swarm = MonteCarloSwarm(agents=[agent1, agent2], iterations=100, aggregator=aggregate_weighted_vote)


# # If using weighted voting, you'll need to adjust the aggregator call to provide the weights:
# # weights = list(range(100, 0, -1))  # Example weights for 100 iterations
# # swarm = MonteCarloSwarm(agents=[agent1, agent2], iterations=100, aggregator=lambda results: aggregate_weighted_vote(results, weights))
```