```
import asyncio

import os

import threading

from concurrent.futures import ThreadPoolExecutor

from dataclasses import dataclass

from multiprocessing import cpu_count

from typing import Any, List


import psutil


from swarms.structs.agent import Agent

from swarms.structs.omni_agent_types import AgentType

from swarms.utils.wrapper_clusterop import (

    exec_callable_with_clusterops,

)



def run_single_agent(agent: AgentType, task: str) -> Any:

    """Run a single agent synchronously"""

    return agent.run(task)



async def run_agent_async(

    agent: AgentType, task: str, executor: ThreadPoolExecutor

) -> Any:

    """
    Run an agent asynchronously using a thread executor.

    Args:
        agent: Agent instance to run
        task: Task string to execute
        executor: ThreadPoolExecutor instance for handling CPU-bound operations

    Returns:
        Agent execution result
    """
    loop = asyncio.get_event_loop()
    return await loop.run_in_executor(
        executor, run_single_agent, agent, task
    )


async def run_agents_concurrently_async(
    agents: List[AgentType], task: str, executor: ThreadPoolExecutor
) -> List[Any]:
    """
    Run multiple agents concurrently using asyncio and thread executor.

    Args:
        agents: List of Agent instances to run concurrently
        task: Task string to execute
        executor: ThreadPoolExecutor for CPU-bound operations
    Returns:
        List of outputs from each agent
    """
    results = await asyncio.gather(
        *(run_agent_async(agent, task, executor) for agent in agents)
    )
    return results


def run_agents_concurrently(
    agents: List[AgentType],
    task: str,
    batch_size: int = None,
    max_workers: int = None,
) -> List[Any]:
    """
    Optimized concurrent agent runner using both uvloop and ThreadPoolExecutor.

    Args:
        agents: List of Agent instances to run concurrently
        task: Task string to execute
        batch_size: Number of agents to run in parallel in each batch (defaults to CPU count)
        max_workers: Maximum number of threads in the executor (defaults to CPU count * 2)

    Returns:
```

```
    List of outputs from each agent
"""
# Optimize defaults based on system resources
cpu_cores = cpu_count()
batch_size = batch_size or cpu_cores
max_workers = max_workers or cpu_cores * 2

results = []

# Get or create event loop
try:
    loop = asyncio.get_event_loop()
except RuntimeError:
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)

# Create a shared thread pool executor with optimal worker count
with ThreadPoolExecutor(max_workers=max_workers) as executor:
    # Process agents in batches
    for i in range(0, len(agents), batch_size):
        batch = agents[i : i + batch_size]
        batch_results = loop.run_until_complete(
            run_agents_concurrently_async(batch, task, executor)
        )
        results.extend(batch_results)
    return results




def run_agents_concurrently_multiprocess(
    agents: List[Agent], task: str, batch_size: int = cpu_count()
) -> List[Any]:
    """
    Manage and run multiple agents concurrently in batches, with optimized performance.

    Args:
        agents (List[Agent]): List of Agent instances to run concurrently.
        task (str): The task string to execute by all agents.
        batch_size (int, optional): Number of agents to run in parallel in each batch.
                                    Defaults to the number of CPU cores.

    Returns:
        List[Any]: A list of outputs from each agent.
    """
    results = []
    loop = asyncio.get_event_loop()

    # Process agents in batches to avoid overwhelming system resources
    for i in range(0, len(agents), batch_size):
        batch = agents[i : i + batch_size]
        batch_results = loop.run_until_complete(
            run_agents_concurrently_async(batch, task)
        )
        results.extend(batch_results)

    return results


def run_agents_sequentially(
    agents: List[AgentType], task: str
) -> List[Any]:
    """
    Run multiple agents sequentially for baseline comparison.

    Args:
        agents: List of Agent instances to run
        task: Task string to execute

    Returns:
        List of outputs from each agent
    """
    return [run_single_agent(agent, task) for agent in agents]


def run_agents_with_different_tasks(
    agent_task_pairs: List[tuple[AgentType, str]],
    batch_size: int = None,
    max_workers: int = None,
) -> List[Any]:
    """
    Run multiple agents with different tasks concurrently.

    Args:
        agent_task_pairs: List of (agent, task) tuples
        batch_size: Number of agents to run in parallel
        max_workers: Maximum number of threads

    Returns:
        List of outputs from each agent
    """

    async def run_pair_async(
        pair: tuple[AgentType, str], executor: ThreadPoolExecutor
    ) -> Any:
        agent, task = pair
        return await run_agent_async(agent, task, executor)

    cpu_cores = cpu_count()
    batch_size = batch_size or cpu_cores
    max_workers = max_workers or cpu_cores * 2
    results = []

    try:
        loop = asyncio.get_event_loop()
    except RuntimeError:
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for i in range(0, len(agent_task_pairs), batch_size):
            batch = agent_task_pairs[i : i + batch_size]
            batch_results = loop.run_until_complete(
                asyncio.gather(
                    *(
                        run_pair_async(pair, executor)
                        for pair in batch
                    )
                )
            )
            results.extend(batch_results)

    return results


async def run_agent_with_timeout(
    agent: AgentType,
    task: str,
    timeout: float,
    executor: ThreadPoolExecutor,
) -> Any:
    """
    Run an agent with a timeout limit.

    Args:
        agent: Agent instance to run
        task: Task string to execute
        timeout: Timeout in seconds
        executor: ThreadPoolExecutor instance

    Returns:
        Agent execution result or None if timeout occurs
    """
    try:
        return await asyncio.wait_for(
            run_agent_async(agent, task, executor), timeout=timeout
        )
    except asyncio.TimeoutError:
        return None


def run_agents_with_timeout(
    agents: List[AgentType],
    task: str,
    timeout: float,
    batch_size: int = None,
    max_workers: int = None,
) -> List[Any]:
    """
    Run multiple agents concurrently with a timeout for each agent.

    Args:
        agents: List of Agent instances
        task: Task string to execute
        timeout: Timeout in seconds for each agent
        batch_size: Number of agents to run in parallel
        max_workers: Maximum number of threads

    Returns:
        List of outputs (None for timed out agents)
    """
    cpu_cores = cpu_count()
    batch_size = batch_size or cpu_cores
    max_workers = max_workers or cpu_cores * 2
    results = []

    try:
        loop = asyncio.get_event_loop()
    except RuntimeError:
        loop = asyncio.new_event_loop()
        asyncio.set_event_loop(loop)

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for i in range(0, len(agents), batch_size):
            batch = agents[i : i + batch_size]
            batch_results = loop.run_until_complete(
                asyncio.gather(
                    *(
                        run_agent_with_timeout(
                            agent, task, timeout, executor
                        )
                        for agent in batch
                    )
                )
            )
            results.extend(batch_results)

    return results



@dataclass
class ResourceMetrics:
    cpu_percent: float
    memory_percent: float
    active_threads: int



def get_system_metrics() -> ResourceMetrics:
    """Get current system resource usage"""
    return ResourceMetrics(
        cpu_percent=psutil.cpu_percent(),
        memory_percent=psutil.virtual_memory().percent,
        active_threads=threading.active_count(),
    )



def run_agents_with_resource_monitoring(
    agents: List[AgentType],
    task: str,
    cpu_threshold: float = 90.0,
    memory_threshold: float = 90.0,
    check_interval: float = 1.0,
) -> List[Any]:
    """
    Run agents with system resource monitoring and adaptive batch sizing.

    Args:
        agents: List of Agent instances
        task: Task string to execute
        cpu_threshold: Max CPU usage percentage
        memory_threshold: Max memory usage percentage
        check_interval: Resource check interval in seconds

    Returns:
        List of outputs from each agent
    """

    async def monitor_resources():
        while True:
            metrics = get_system_metrics()
            if (
                metrics.cpu_percent > cpu_threshold
                or metrics.memory_percent > memory_threshold
            ):
                # Reduce batch size or pause execution
                pass
            await asyncio.sleep(check_interval)

    # Implementation details...


def _run_agents_with_tasks_concurrently(
    agents: List[AgentType],
    tasks: List[str] = [],
    batch_size: int = None,
    max_workers: int = None,
) -> List[Any]:
    """
    Run multiple agents with corresponding tasks concurrently.

    Args:
```

```
    agents: List of Agent instances to run
    tasks: List of task strings to execute
    batch_size: Number of agents to run in parallel
    max_workers: Maximum number of threads

Returns:
    List of outputs from each agent
"""
if len(agents) != len(tasks):
    raise ValueError(
        "The number of agents must match the number of tasks."
    )

cpu_cores = os.cpu_count()
batch_size = batch_size or cpu_cores
max_workers = max_workers or cpu_cores * 2
results = []

try:
    loop = asyncio.get_event_loop()
except RuntimeError:
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)

async def run_agent_task_pair(
    agent: AgentType, task: str, executor: ThreadPoolExecutor
    ) -> Any:
        return await run_agent_async(agent, task, executor)

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        for i in range(0, len(agents), batch_size):
            batch_agents = agents[i : i + batch_size]
            batch_tasks = tasks[i : i + batch_size]
            batch_results = loop.run_until_complete(
                asyncio.gather(
                    *(
                        run_agent_task_pair(agent, task, executor)
                        for agent, task in zip(
                            batch_agents, batch_tasks
                        )
                    )
                )
            )
            results.extend(batch_results)

    return results


def run_agents_with_tasks_concurrently(
    agents: List[AgentType],
    tasks: List[str] = [],
    batch_size: int = None,
    max_workers: int = None,
    device: str = "cpu",
    device_id: int = 1,
    all_cores: bool = True,
    no_clusterops: bool = False,
) -> List[Any]:
    """
    Executes a list of agents with their corresponding tasks concurrently on a specified device.

    This function orchestrates the concurrent execution of a list of agents with their respective tasks on a specified device, either CPU or GPU. It leverages the `exec_callable_with_clusterops` function to manage the execution on the specified device.

    Args:
        agents (List[AgentType]): A list of Agent instances or callable functions to execute concurrently.
        tasks (List[str], optional): A list of task strings to execute for each agent. Defaults to an empty list.
        batch_size (int, optional): The number of agents to run in parallel. Defaults to None.
        max_workers (int, optional): The maximum number of threads to use for execution. Defaults to None.
        device (str, optional): The device to use for execution. Defaults to "cpu".
        device_id (int, optional): The ID of the GPU to use if device is set to "gpu". Defaults to 0.
        all_cores (bool, optional): If True, uses all available CPU cores. Defaults to True.

    Returns:
        List[Any]: A list of outputs from each agent execution.
    """
    # Make the first agent not use the ifrs

    if no_clusterops:
        return _run_agents_with_tasks_concurrently(
            agents, tasks, batch_size, max_workers
        )
    else:
        return exec_callable_with_clusterops(
            device,
            device_id,
            all_cores,
            _run_agents_with_tasks_concurrently,
            agents,
            tasks,
            batch_size,
            max_workers,
        )


# # Example usage:
# # Initialize your agents with the same model to avoid re-creating it
# agents = [
#     Agent(
#         agent_name=f"Financial-Analysis-Agent_parallel_swarm{i}",
#         system_prompt=FINANCIAL_AGENT_SYS_PROMPT,
#         llm=model,
#         max_loops=1,
#         autosave=True,
#         dashboard=False,
#         verbose=False,
#         dynamic_temperature_enabled=False,
#         saved_state_path=f"finance_agent_{i}.json",
#         user_name="swarms_corp",
#         retry_attempts=1,
#         context_length=200000,
#         return_step_meta=False,
#     )
#     for i in range(5)  # Assuming you want 10 agents
# ]

# task = "How can I establish a ROTH IRA to buy stocks and get a tax break? What are the criteria"
# outputs = run_agents_concurrently(agents, task)

# for i, output in enumerate(outputs):
#     print(f"Output from agent {i+1}:\n{output}")
```