Task Graph Executor - API Documentation

Table of Contents

Core Classes

Task

Represents a single executable unit with dependencies.

```
@dataclass
class Task:
    name: str                    # Unique identifier
    func: Callable               # Function to execute
    dependencies: List[str] = []    # Task names this depends on
    args: tuple = ()             # Positional arguments
    kwargs: dict = {}            # Keyword arguments
    retries: int = 0             # Retry attempts on failure
    timeout: Optional[float] = None # Execution timeout
```

Example:

```
task = Task(
    name="process_data",
    func=process_function,
    dependencies=["fetch_data"],
    args=(input_data,),
    kwargs={"mode": "fast"},
    retries=3
)
```

TaskGraphExecutor

Main executor class that orchestrates task execution.

```
executor = TaskGraphExecutor(max_workers=4)
```

Parameters:

- `max_workers` (int): Maximum number of concurrent tasks (default: 4)

API Reference

Adding Tasks

`add_task(task: Task)`

Add a task to the execution graph.

executor.add_task(Task("compile", compile_code))
executor.add_task(Task("test", run_tests, dependencies=["compile"]))

Raises:

- **`ValueError`**: If task with same name already exists

Validation

`validate_graph()`

Validate the task graph for cycles and missing dependencies.

executor.validate_graph()

Raises:

- `CyclicDependencyError`: If circular dependency detected
- `ValueError`: If dependency references non-existent task

Implementation: Uses Depth-First Search (DFS) with recursion stack to detect cycles in O(V + E) time.

Execution

`execute() -> Dict[str, Any]`

Execute all tasks in the graph.

results = executor.execute()
# Returns: {"task_name": result, ...}

Behavior:

1. Validates graph for cycles
2. Executes tasks in topological order
3. Maximizes parallelism by running independent tasks simultaneously
4. Handles failures and retries
5. Returns results dictionary

Returns: Dictionary mapping task names to their return values

Topological Sorting

`topological_sort() -> List[str]`

Returns tasks in dependency order.

```
order = executor.topological_sort()
# Returns: ["task_a", "task_b", "task_c"]
```

Implementation: Kahn's algorithm with in-degree tracking

Statistics

`get_execution_stats() -> Dict[str, Any]`

Get detailed execution statistics.

```
stats = executor.get_execution_stats()
```

Returns:

```
{
    "total_tasks": int,
    "completed": int,
    "failed": int,
    "running": int,
    "task_details": {
        "task_name": {
            "status": str,
            "duration": float,
            "retries": int,
            "error": Optional[str]
        }
    }
}
```

Visualization

`visualize_graph() -> str`

Generate ASCII representation of task graph.

```
print(executor.visualize_graph())
```

Output:

```
Task Dependency Graph:
============================================================
compile_main (no dependencies)
compile_utils (no dependencies)
link <- ['compile_main', 'compile_utils']
```

```
test <- ['link']
```

Configuration

Thread Pool Sizing

```
# CPU-bound tasks
executor = TaskGraphExecutor(max_workers=cpu_count())

# I/O-bound tasks
executor = TaskGraphExecutor(max_workers=cpu_count() * 2)

# Mixed workload
executor = TaskGraphExecutor(max_workers=8)
```

Retry Configuration

```
task = Task(
    "flaky_api_call",
    api_function,
    retries=3  # Will attempt up to 4 times total
)
```

Retry Behavior:

- Immediate retry (1 second delay)
- Exponential backoff not implemented (can be added)
- Final failure marks task as FAILED

Examples

Example 1: Build System

```
executor = TaskGraphExecutor(max_workers=4)

# Compile modules in parallel
executor.add_task(Task("compile_main", compile, args=("main.c",)))
executor.add_task(Task("compile_utils", compile, args=("utils.c",)))
executor.add_task(Task("compile_config", compile, args=("config.c",)))

# Link after all compilation completes
executor.add_task(Task(
    "link",
    link_objects,
    dependencies=["compile_main", "compile_utils", "compile_config"]
))

# Test the binary
```

```
executor.add_task(Task("test", run_tests, dependencies=["link"]))

results = executor.execute()
```

Example 2: Data Pipeline

```
executor = TaskGraphExecutor(max_workers=3)

# Extract from multiple sources
executor.add_task(Task("extract_users", extract_data, args=("users_db",)))
executor.add_task(Task("extract_orders", extract_data, args=("orders_db",)))

# Transform extracted data
executor.add_task(Task(
    "transform_users",
    transform,
    args=("normalize",),
    dependencies=["extract_users"]
))

executor.add_task(Task(
    "transform_orders",
    transform,
    args=("aggregate",),
    dependencies=["extract_orders"]
))

# Load to warehouse
executor.add_task(Task(
    "load_warehouse",
    load_data,
    dependencies=["transform_users", "transform_orders"]
))

results = executor.execute()
```

Example 3: With Visualization

```
from task_graph_visualization import VisualizationExecutor

executor = VisualizationExecutor(max_workers=4, monitoring_port=8090)

# Add tasks...
executor.add_task(Task("step1", process))
executor.add_task(Task("step2", validate, dependencies=["step1"]))

# Start monitoring server
executor.start_monitoring()
print("Dashboard: http://localhost:8090")

# Execute with live monitoring
results = executor.execute()
```

```
# Keep server running
input("Press Enter to stop...")
executor.stop_monitoring()
```

Error Handling

Task Failures

When a task fails:

1. Task status set to FAILED
2. Error captured in `task.error`
3. Dependent tasks marked as SKIPPED
4. Execution continues for independent tasks

```
try:
    results = executor.execute()
except Exception:
    # Graph validation failed
    pass

# Check individual task failures
stats = executor.get_execution_stats()
for name, details in stats['task_details'].items():
    if details['status'] == 'FAILED':
        print(f"{name} failed: {details['error']}")
```

Common Exceptions

Exception	Cause	Solution
`CyclicDependencyError`	Circular dependencies detected	Review task dependencies
`ValueError`	Missing dependency or duplicate name	Check task names and dependencies
`Exception` (from task)	Task execution failed	Check task function implementation

Debugging Failed Executions

```
# Get detailed stats
stats = executor.get_execution_stats()

# Check failed tasks
failed = [name for name, details in stats['task_details'].items()
          if details['status'] == 'FAILED']

# Inspect specific task
task = executor.tasks['failed_task_name']
print(f"Error: {task.error}")
print(f"Retries: {task.retry_count}")
print(f"Duration: {task.duration()}")
```

Best Practices

1. Task Granularity

Good: Fine-grained tasks enable better parallelism

```
executor.add_task(Task("download", download_file))
executor.add_task(Task("parse", parse_file, dependencies=["download"]))
executor.add_task(Task("validate", validate, dependencies=["parse"]))
```

Bad: Coarse-grained tasks limit parallelism

```
executor.add_task(Task("do_everything", download_parse_validate))
```

2. Dependency Management

Good: Explicit, minimal dependencies

```
# Task C only needs A, not B
executor.add_task(Task("C", func_c, dependencies=["A"]))
```

Bad: Unnecessary dependencies

```
# Task C depends on B unnecessarily
executor.add_task(Task("C", func_c, dependencies=["A", "B"]))
```

3. Worker Configuration

```
# For CPU-bound tasks (computation)
max_workers = os.cpu_count()

# For I/O-bound tasks (network, disk)
max_workers = os.cpu_count() * 2
```

```
# For mixed workloads
max_workers = os.cpu_count() + 2
```

4. Error Handling

```
def robust_task():
    try:
        result = risky_operation()
        return result
    except SpecificError as e:
        # Handle expected errors
        return default_value
    # Let unexpected errors propagate for retry
```

5. Idempotency

Design tasks to be idempotent (can be safely retried):

```
def idempotent_task(output_path):
    # Check if already done
    if os.path.exists(output_path):
        return load_existing(output_path)

    # Do work
    result = process_data()
    save_result(result, output_path)
    return result
```

6. Logging

```
import logging

def logged_task(data):
    logging.info(f"Processing {len(data)} items")
    result = process(data)
    logging.info(f"Completed with {len(result)} results")
    return result
```

Performance Considerations

Time Complexity

Operation	Complexity	Notes
Add Task	O(1)	Amortized
Validate Graph	O(V + E)	DFS traversal

Topological Sort	O(V + E)	Kahn's algorithm
Get Ready Tasks	O(V)	Check all tasks
Execute	O(V + E)	Plus actual task execution time

Where V = number of tasks, E = number of dependencies

Space Complexity

- Task storage: O(V)
- Dependency graphs: O(V + E)
- Thread pool: O(max_workers)
- Total: O(V + E + max_workers)

Optimization Tips

1. **Minimize dependency chains**: Deeper chains reduce parallelism
2. **Balance worker count**: Too few = underutilized, too many = overhead
3. **Use retries sparingly**: Only for transient failures
4. **Profile task duration**: Identify bottlenecks in critical path

---

Thread Safety

The executor is thread-safe for:

- Adding tasks (before execution)
- Concurrent task execution
- Statistics retrieval

Not thread-safe for:

- Adding tasks during execution
- Modifying task properties during execution

```
# Safe
executor.add_task(task1)
executor.add_task(task2)
results = executor.execute()

# Unsafe
def background_add():
    executor.add_task(new_task)  # Don't do this during execution

executor.execute()
```

Limitations

1. **No distributed execution**: Single-machine only
2. **No task priority**: All tasks at same level have equal priority
3. **No dynamic dependencies**: Dependencies must be known upfront
4. **No resource constraints**: Doesn't track memory, CPU, etc.
5. **Simple retry**: No exponential backoff or jitter

Future Enhancements

Potential improvements for production use:

- Distributed execution across multiple machines
- Priority-based scheduling
- Resource constraints (memory, CPU limits)
- Advanced retry strategies (exponential backoff)
- Persistent state for resuming failed executions
- Task result caching
- Dynamic task generation
- Integration with job queues (Celery, RQ)

Appendix: Algorithm Details

Cycle Detection (DFS)

```
def has_cycle(node, visited, recursion_stack):
    visited.add(node)
    recursion_stack.add(node)

    for neighbor in dependencies[node]:
        if neighbor not in visited:
            if has_cycle(neighbor, visited, recursion_stack):
                return True
        elif neighbor in recursion_stack:
            return True  # Back edge found = cycle

    recursion_stack.remove(node)
    return False
```

Topological Sort (Kahn's Algorithm)

```
def topological_sort():
    in_degree = {task: len(dependencies[task]) for task in tasks}
    queue = [task for task, degree in in_degree.items() if degree == 0]
    sorted_tasks = []

    while queue:
        task = queue.pop(0)
```

```
        sorted_tasks.append(task)

        for dependent in dependents[task]:
            in_degree[dependent] -= 1
            if in_degree[dependent] == 0:
                queue.append(dependent)

    return sorted_tasks
```

For additional examples and use cases, see the test suite in `task_graph_tests.py`.