Algorithm Project

Task Allocation and Scheduling in Distributed Systems using Extended Minimum Cost Maximum Flow and Dynamic Programming

Designed by Sayin Ala

1. Introduction

1.1 Purpose of the Project

In modern distributed computing environments, efficient allocation of computational tasks to available resources is critical for achieving optimal system performance, cost-efficiency, and responsiveness. As distributed systems scale in complexity and heterogeneity, traditional task assignment and scheduling approaches often fall short in addressing the multi-dimensional constraints such as resource limitations, execution costs, task dependencies, and time deadlines.

This project presents a comprehensive, multi-phase framework for task allocation and scheduling in distributed systems, combining the power of **Minimum Cost Maximum Flow (MCMF)** algorithms with the flexibility of **Dynamic Programming (DP)**. The core objective is to minimize the total execution and communication cost while respecting computational constraints and maintaining task execution feasibility.

The proposed framework models the distributed environment as a dynamic flow network, where tasks are treated as demand nodes and computing resources as supply nodes. Through a series of phases, the system handles not only static allocation but also incorporates time-aware scheduling, dependency constraints, and real-time reallocation in response to runtime changes such as node failures or new task arrivals.

By integrating MCMF for global optimization and DP for localized decision-making at the node level, the system achieves a balance between scalability and adaptability. This document outlines the phased development of the solution, presents corresponding test scenarios and data, and demonstrates how the proposed methodology can effectively improve task throughput, cost-efficiency, and system resilience in distributed computing environments.

1.2 Key Features

The application will include the following core features:

🔄 Multi-Phase Architecture

The project is structured into four well-defined phases, each addressing a specific aspect of distributed task allocation and scheduling—from initial static allocation to dynamic runtime adjustments.

📊 Graph-Based Task Allocation Using MCMF

Tasks and nodes are modeled in a flow network where task-to-node assignments are optimized using the **Minimum Cost Maximum Flow** algorithm, ensuring minimum execution and communication cost while respecting resource constraints.

⏳ Time-Aware Scheduling

Through **time-expanded graphs** or auxiliary scheduling mechanisms, the framework incorporates task deadlines, availability windows, and execution durations, making the allocation sensitive to temporal constraints.

🔗 Support for Task Dependencies

The system models and respects inter-task dependencies (e.g., task A must complete before task B starts), which are integrated either into the flow graph or resolved via scheduling logic.

⚙️ Dynamic Reconfiguration

The system can react to runtime events such as:

- **Node failures**
- **New task arrivals**
- **Resource fluctuations**

These are handled through **incremental reoptimization** of the flow, minimizing disruption and reallocation costs.

📌 Local Scheduling via Dynamic Programming

Once tasks are assigned to nodes, a **DP-based local scheduler** determines the optimal execution order for tasks on each node, considering available CPU slots and task durations.

📐 Customizable Test Framework

Includes a modular test suite for each phase:

- Task generation
- Resource modeling
- Scenario injection (failures, dependencies, time constraints)
  This allows controlled experimentation and benchmarking.

🧠 Modular and Extensible Design

The framework is designed to allow:

- Plug-in optimization strategies (e.g., heuristic, ML-assisted flow adjustment)
- Variable cost models (energy, latency, priority)
- Integration with real-world schedulers and simulators

📊 Performance-Oriented

The architecture is built with performance in mind:

- Efficient graph traversal and flow algorithms
- Low overhead in incremental updates
- Scalability to hundreds of tasks and nodes

Phase 1: Initial Task-to-Node Allocation using Minimum Cost Maximum Flow (MCMF)

- Overview

The first phase of the project focuses on assigning computational tasks to distributed nodes in an optimal manner, using a flow-based approach. The primary goal is to ensure that all tasks are assigned to eligible computing nodes while minimizing the total execution cost under resource constraints.

This phase uses the **Minimum Cost Maximum Flow (MCMF)** algorithm to model the task allocation problem as a flow network, where tasks act as demand sources and computing nodes act as processing sinks. This transformation allows us to take advantage of efficient graph-based flow algorithms to handle complex, multi-dimensional optimization.

- Problem Modeling

We define the system as a directed flow graph G(V,E) with the following components:

- **Source Node (S):** The origin of task flows.
- **Task Nodes (T_i):** Each representing a computational task that must be executed once.
- **Compute Nodes (N_j):** Each representing a distributed node with limited CPU and memory resources.
- **Sink Node (K):** Collects the final flow from compute nodes to enforce complete assignment.

Each edge in the graph carries:

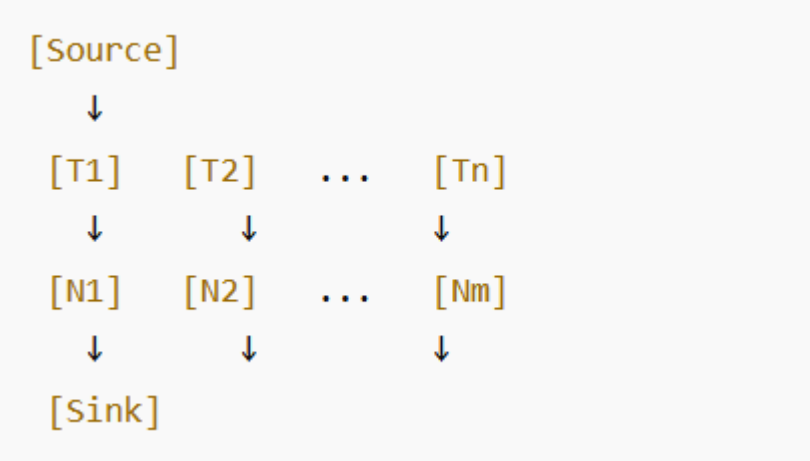
- **Capacity:** Represents resource limitations (e.g., a task can be assigned only once; a node can accept multiple tasks based on capacity).
- **Cost:** Represents the execution cost of a task on a given node.

- Constraints Modeled in Graph

- **Resource Constraints:** Tasks are only assignable to nodes that can satisfy their CPU and memory needs.
- **Execution Cost:** Each task-node edge is assigned a cost based on task-specific computational requirements on that node.
- **One-to-One Assignment:** Each task must be assigned to exactly one node (unit capacity from task to node).
- **Node Capacity:** Each compute node has a total CPU/memory budget across all assigned tasks.

- Flow Graph Structure

Example structure:



- Edges from Source → Task nodes: capacity = 1, cost = 0
- Edges from Task → Node: capacity = 1, cost = execution cost
- Edges from Node → Sink: capacity = node's total task limit, cost = 0

- Algorithm Used

The algorithm implemented for this phase is typically one of the following:

- **Successive Shortest Path Algorithm**
- **Cycle Canceling**
- **Cost-Scaling Push-Relabel**

These ensure that:

- The maximum number of tasks are assigned (maximum flow)
- The total execution cost is minimized (minimum cost)

- Advantages of This Model

- Supports any number of tasks and nodes
- Naturally handles heterogeneity in execution costs
- Can be extended to support time and failure constraints (in future phases)

- Input and Output of Phase 1

◇ Input

- **List of Tasks**, each with:
  - id, cpu, ram, deadline
- **List of Nodes**, each with:
  - id, cpu_capacity, ram_capacity
- **Execution Cost Matrix**:
  - cost[task_id][node_id] = cost value or ∞ if not executable

📥 Example (JSON-like structure):

```
{
 "tasks": [
  {"id": "T1", "cpu": 2, "ram": 4, "deadline": 2},
  {"id": "T2", "cpu": 1, "ram": 2, "deadline": 3}
 ],
 "nodes": [
  {"id": "N1", "cpu_capacity": 5, "ram_capacity": 6},
  {"id": "N2", "cpu_capacity": 3, "ram_capacity": 3}
 ],
 "exec_cost": {
  "T1": {"N1": 4, "N2": 6},
  "T2": {"N1": 3, "N2": 2}
 }
}
```

◇ Output

- **Task Assignments:** A list showing which task is assigned to which node
- **Total Cost:** Sum of execution costs of all assignments

📤 Example:

```
{
 "assignments": {
  "T1": "N1",
  "T2": "N2"
 },
 "total_cost": 6
}
```

Phase 2: Time-Aware Scheduling and Dependency Handling in Extended Flow Graph

- Overview

In Phase 1, task-to-node assignment was optimized based on cost and capacity using the Minimum Cost Maximum Flow algorithm. However, real-world distributed systems often operate under **time constraints** and **execution dependencies** between tasks.

Phase 2 addresses these challenges by incorporating **temporal scheduling** and **task dependency modeling** into the resource allocation framework. This is done by extending the static flow graph into a **time-expanded flow network** or by supplementing MCMF with additional **post-processing scheduling logic**.

- Objectives

  - Ensure tasks are **scheduled within allowed time windows**, respecting **deadlines**.
  - Enforce **task dependencies**, e.g., Task A must complete before Task B starts.
  - Honor **node-level resource availability per time slot**.
  - Preserve the efficiency of task assignments from Phase 1 while validating them against the temporal constraints.

- Design Approach

There are two primary ways to integrate scheduling constraints into the existing MCMF model:

1. Time-Expanded Flow Network

  - Each task is replicated across possible **time slots** (e.g., T1@t0, T1@t1, T1@t2...).
  - Edges from tasks to compute nodes are now timestamped to reflect start times.
  - **Dependency constraints** are modeled as directed edges enforcing precedence across time slots.

This method enables a **purely MCMF-based solution** but at the cost of increased graph size.

2. Two-Stage Model: Allocation + Scheduling

  - First, run the Phase 1 MCMF to assign tasks to nodes.
  - Then, for each node:

    - Use **greedy or dynamic programming** to build a valid local execution schedule across time.
    - Enforce dependencies between tasks through **topological sorting** or **backtracking**.

This method is more **modular and scalable**, especially for systems with limited time granularity.

- Constraints Introduced in Phase 2

    - **Deadlines:** Each task must be completed before its deadline (latest finish time).
    - **Start Times:** Tasks may have earliest start times or availability windows.
    - **Dependencies:** A directed acyclic graph (DAG) defines task precedence.
    - **Time-Sloted Node Capacities:** Each compute node can only handle a certain amount of CPU/RAM per time slot.

- Scheduling Strategy

Depending on the method:

- If using **time-expanded graph**, MCMF determines both assignment and schedule.
- If using **post-MCMF scheduling**, apply:
    - **Topological sort** for ordering dependent tasks.
    - **Greedy or DP** to fit tasks into time slots based on node availability and resource limits.

- Benefits of Phase 2

    - Ensures **feasibility** of execution beyond just static assignment.
    - Brings model closer to **real-world constraints** (e.g., workflows, deadlines).
    - Keeps flow-based allocation extensible and relevant in dynamic time contexts.

- Input and Output of Phase 2

    - Input

Extends Phase 1 with the following additional data:

- **Time Slots:** A list of available time units (e.g., [0, 1, 2, 3, 4])
- **Node Resource Availability per Time Slot:**
  For each node and time slot, define available CPU/RAM.
- **Task Dependencies:**
  A list of task relationships where one must finish before another can start.
- **Task Duration (Optional):**
  Duration in time units each task takes to execute.

Example (JSON-like structure):

```
{
  "tasks": [
    {
      "id": "T1",
      "cpu": 2,
      "ram": 4,
      "deadline": 3
    },
    {
      "id": "T2",
      "cpu": 1,
      "ram": 2,
      "deadline": 3
    },
    {
      "id": "T3",
      "cpu": 3,
      "ram": 3,
      "deadline": 4
    }
  ],
  "nodes": [
    {
      "id": "N1",
      "cpu_capacity": 5,
      "ram_capacity": 6
    },
    {
      "id": "N2",
      "cpu_capacity": 6,
      "ram_capacity": 5
    }
  ],
  "exec_cost": {
    "T1": {
      "N1": 4,
      "N2": 2
    },
    "T2": {
```

```
            "N1": 3,
            "N2": 4
        },
        "T3": {
            "N1": 2,
            "N2": 3
        }
    },
    "time_slots": [
        0,
        1,
        2,
        3
    ],
    "node_capacity_per_time": {
        "N1": {
            "0": 2,
            "1": 2,
            "2": 2,
            "3": 2
        },
        "N2": {
            "0": 3,
            "1": 3,
            "2": 2,
            "3": 2
        }
    },
    "dependencies": [
        {
            "before": "T1",
            "after": "T3"
        },
        {
            "before": "T2",
            "after": "T3"
        }
    ],
    "task_duration": {
        "T1": 1,
        "T2": 1,
        "T3": 2
    }
}
```


- Output

The output should specify a **valid schedule** for each task, respecting:

- Assignment to a node
- Start time (within the time slot limits)
- Dependency constraints
- Deadline satisfaction

📤 Example:

```
{
  "schedule": {
    "T1": {"node": "N1", "start_time": 0},
    "T2": {"node": "N2", "start_time": 1},
    "T3": {"node": "N1", "start_time": 2}
  },
  "valid": true,
  "total_cost": 9
}
```

If scheduling fails due to infeasibility (e.g. a deadline conflict), the `valid` flag would be set to `false`.

Phase 3: Dynamic Reallocation in Response to Runtime Events

- Overview

While Phases 1 and 2 ensure optimal task allocation and scheduling under static conditions, real-world distributed systems are subject to **dynamic changes** such as:

- Node failures
- Arrival of new tasks
- Changes in resource availability
- Delays or slow execution on certain nodes

Phase 3 introduces **dynamic reallocation and reoptimization** mechanisms to adapt the system's task schedule with minimal disruption and cost. The goal is to **update the current assignment and schedule efficiently**, without re-running the entire MCMF from scratch.

- Objectives

- React to system changes **in real time**.
- **Reallocate tasks** only when necessary, preserving as much of the previous schedule as possible.
- Maintain system constraints (deadlines, capacities, dependencies).
- Minimize **reallocation cost**, which may include:
    - Task migration overhead
    - Reconfiguration latency
    - Change penalty (e.g. minimizing the number of modified assignments)

- Types of Dynamic Events Handled

1. Node Failure

A compute node becomes unavailable at a certain time, requiring immediate reassignment of its scheduled tasks.

2. New Task Arrival

A new task is added to the system mid-execution, needing on-the-fly assignment and scheduling.

3. Capacity Change

A node's available resources change dynamically due to system load or administrative decisions.

- Reallocation Strategy

  - **Incremental Flow Update:** Re-run the MCMF only for affected portions of the flow network (e.g. remove failed node edges, add new task edges).
  - **Partial Rescheduling:** Apply localized re-scheduling using greedy or DP for only the changed set of tasks.
  - **Cost-Aware Reassignment:** Minimize changes to the previous schedule to avoid high migration or disruption cost.

- Graph Update Operations

  - Remove failed nodes and their incident edges.
  - Add new task nodes and connect them to eligible compute nodes.
  - Recalculate residual capacities and costs.
  - Optionally, include **reconfiguration penalties** in edge costs to discourage excessive reassignment.

- Benefits of Phase 3

  - Enhances system **resilience and flexibility**
  - Reduces **downtime** and **wasted computation**
  - Enables **real-time adaptation** without full recomputation
  - Supports **on-demand scaling** and task injection

- Input and Output of Phase 3

  - Input

Includes all inputs from **Phases 1 and 2**, plus a list of dynamic runtime events.

Example (JSON-like structure):

```
{
 "previous_schedule": {
  "T1": {"node": "N1", "start_time": 0},
  "T2": {"node": "N2", "start_time": 1},
  "T3": {"node": "N1", "start_time": 2}
 },
```

```
 "events": [
  {
   "type": "node_failure",
   "node": "N2",
   "time": 1
  },
  {
   "type": "new_task",
   "task": {
    "id": "T4",
    "cpu": 2,
    "ram": 2,
    "deadline": 4,
    "duration": 1,
    "exec_cost": {
     "N1": 3,
     "N3": 2
    }
   }
  }
 ],
 "node_capacity_update": {
  "N1": {"2": 1},  // CPU reduced at time slot 2
  "N3": {"1": 2}   // CPU added at time slot 1
 }
}
```

◇ Output

The output will be an **updated task schedule**, reflecting the changes and a measure of disruption:

Example:

```
{
 "updated_schedule": {
  "T1": {"node": "N1", "start_time": 0},
  "T3": {"node": "N1", "start_time": 2},
  "T4": {"node": "N3", "start_time": 1}
 },
 "reassigned_tasks": ["T2", "T4"],
 "failed_tasks": ["T2"],  // T2 could not be reassigned in time
 "total_cost": 11,
 "change_penalty": 2
}
```

- Explanation:
    - reassigned_tasks: tasks that were migrated or newly scheduled.
    - failed_tasks: tasks that couldn't meet deadline or resource constraints post-reallocation.
    - change_penalty: cost based on number of changed assignments (e.g., for disruption minimization).

Phase 4: Local Task Scheduling using Dynamic Programming on Nodes

- Overview

After tasks have been globally assigned to nodes (via MCMF) and globally scheduled across time (with respect to dependencies and deadlines), each individual compute node must decide **in what order to execute its assigned tasks**.

Phase 4 handles this **per-node execution scheduling**. Each node may have limited resources per time slot (e.g., CPU threads), and it needs to execute multiple tasks that:

- Vary in duration
- Compete for limited resources
- Have soft or hard deadlines
- Might be prioritized differently

To solve this, we use **Dynamic Programming (DP)** to build a **local task execution schedule** that maximizes efficiency, meets deadlines, and optionally minimizes penalty cost or idle time.

- Objectives

- Assign an execution order and time slot to each task **within the node**
- Ensure resource constraints (CPU, RAM) are satisfied in each time slot
- Meet deadlines for each task, if possible
- Minimize idle resource time or task delay penalty
- Maintain consistency with the global allocation/scheduling results

---

- Problem Model (Per Node)

Each node receives:

- A set of tasks it has been assigned
- The time window in which it is available
- The **CPU/RAM available per time slot**
- Each task's **duration**, **resource need**, and **deadline**

This becomes a **bounded scheduling problem**, which can be modeled similarly to **bounded knapsack** or **interval scheduling with resource constraints**, solvable via DP.

- Scheduling Strategy

State Representation:

dp[time][task_subset] = minimum_penalty_cost

Or, for maximization:

dp[time][task_subset] = maximum_tasks_scheduled

The DP considers:

- All permutations of tasks that fit into the available time and capacity
- Task dependencies (within the node, if any)
- Deadline penalties (optional)
- Preference for earlier completion (optional)

- Benefits of Phase 4

- Fine-tuned control of **in-node scheduling**
- Reduces **resource underutilization** (e.g., idle CPU slots)
- Enables **deadline-aware execution** at the micro-level
- Separates global allocation from node-level execution logic (modular)

- Input and Output of Phase 4

  - Input

Each compute node independently receives:

- A list of assigned tasks
- For each task: CPU needed, RAM needed, duration, and deadline
- A resource profile (available CPU per time slot)
- Optional: deadline penalties, task priorities

📥 Example (JSON-like structure):

```
{
 "node": "N1",
 "assigned_tasks": [
  {"id": "T1", "cpu": 2, "ram": 2, "duration": 1, "deadline": 2},
  {"id": "T3", "cpu": 2, "ram": 1, "duration": 2, "deadline": 4},
  {"id": "T4", "cpu": 1, "ram": 1, "duration": 1, "deadline": 3}
 ],
```

```
  "resource_per_time": {
    "0": {"cpu": 2},
    "1": {"cpu": 2},
    "2": {"cpu": 2},
    "3": {"cpu": 2}
  },
  "time_slots": [0, 1, 2, 3]
}
```

◇ Output

A **per-task execution schedule**, showing:

- Task start time
- Whether the task meets its deadline
- Any penalty or delay

📤 Example:

```
{
  "execution_schedule": {
    "T1": {"start_time": 0, "meets_deadline": true},
    "T4": {"start_time": 1, "meets_deadline": true},
    "T3": {"start_time": 2, "meets_deadline": true}
  },
  "total_idle_time": 0,
  "penalty_cost": 0
}
```

In case of infeasible scheduling due to resource limits or tight deadlines:

```
{
  "execution_schedule": {
    "T1": {"start_time": 0, "meets_deadline": true},
    "T4": {"start_time": 1, "meets_deadline": true},
    "T3": {"start_time": null, "meets_deadline": false}
  },
  "total_idle_time": 1,
  "penalty_cost": 2  // for missing T3's deadline
}
```

- Final Integration

At the end of Phase 4, you will have:

- Global task-to-node assignments (from Phase 1)
- Time-aware scheduling across the system (from Phase 2)
- Adaptation to runtime changes (from Phase 3)
- Locally optimized execution on each node (from Phase 4)