DAG Workflow Scheduling: Design and Analysis

The goal of the workflow scheduling task is to efficiently allocate jobs, considering their dependencies on each other, to a set of homogeneous machines so that their execution is completed in the minimum amount of time. It is a NP-hard problem due to the numerous possible combinations and dependencies involved. Hence, finding an exact solution becomes intractable in the case of large instances of workflow graphs. However, we can employ some heuristics to solve the problem, though such a solution may not guarantee the optimal solution. The algorithm implemented here is broken into two parts:

1. **Finding Execution Order:** Since some jobs are not executable until all the jobs on which they depend are executed and output from them is received, we need to find the correct order of execution. For finding this order, we use a slightly modified Kahn's [1] topological sorting algorithm. Since it is possible that multiple jobs are available for execution at some point in time, which jobs should be executed first in such a situation? One simple heuristic is to give priority to the job that is the most critical, meaning it has the longest execution time following it. So, instead of an ordinary priority queue in Kahn's algorithm, we use a priority queue that gives priority to the most critical job.
2. **Scheduling Job on Machines:** Once we find the order of execution, we need to schedule them on the limited number of machines. Another simple heuristic employed here is that the job is scheduled on the machine, where it will be finished earlier. This approach may not give an optimal solution as it doesn't foresee future communication overhead.

Assumption:

Some conditions necessary for the algorithm designed are not explicitly stated in the problem statement given. So, the following assumptions are made:

- The communication overhead between jobs is only needed when they are executed on different machines. Otherwise, the communication cost between jobs allocated to the same machine is considered to be zero.

Algorithm

Inputs:

- **Workflow Graph:** Task Scheduling DAG represented using adjacency list along with job execution time and communication overhead.
- **Number of Machines:** Number of homogenous independent machine available for parallel job execution.

Step 1: Finding Critical Weight Jobs

- The criticality considers maximum execution and communication times among all paths from the job to the terminal job.
- Use depth first traversal of the graph to find critical weights of all jobs

Step 2: Topological Sorting

- Calculate the indegrees for all jobs in the workflow.
- Initiliaze a empty topological order list
- In a priority queue, insert jobs with zero indegrees. In the priority queue, job with maximum criticality will be in the front/top.
- Until the priority queues is not empty DO:
  - Remove top priority job from the queue and append in the topological order list
  - For each successor of the priority job DO:
    - Reduce indegree of the successor job by 1
    - If indegree of the successor job becomes 0
      - Push the successor job to the priority queue

Step 3: Schedule Workflow

- Initialize variables for machine finish times, job finish times, and a map for job-to-machine assignments.
- Iterate through the topologically sorted jobs.
  - For each job, find the machine that will finish the job earliest by considering communication times and current machine finish times.
  - Update the scheduling information above initialized variable
- Calculate the makespan by finding the maximum finish time among all machines.

Time Complexity Analysis

Let's break down the major operations and analyze their performance individually:

1. **Creating the Worflow DAG**
   - Adding a job in the DAG involves inserting a job object in a hashmap, which takes constant `O(1)` time provided no hash collision occurs.
   - Adding a communication in the DAG involves appending communication links in incoming and outgoing communication adjancency list of the respective job, which also takes constant `O(1)` time.
   - However, since we need to add `V` jobs and `E` communications links, overall DAG creation time complexity becomes `O(V + E)`.
2. **Topological Sort:**
   - Critical weights calculation, calculating maximum amount of time it takes to execute all tasks following the job to the terminal job, is implemented in depth-first search approach. It's time complexity is `O(V + E)` since every jobs and communication links are visited.

    - The topological sort uses a priority queue (heap) to sort the jobs based on their critical weights whose time complexity for pop operation is `O(logV)`.
    - The time complexity of topological sorting than becomes `O(VlogV + E)`. The `logV` term comes from the priority queue operations.
    - Since, sorting is more dominant than the critical weights calculation here, overall time complexity is `O(VlogV + E)`.
3. **Scheduling:**
    - For each job in the topoloical order determined above, scheduling tries to determine which machine will finish the job earliest. Hence, the time complexity is `O(V * K + K * E)`, `K` is the number of machines. This is because, for each job, it may need to check all machines and for each immediate predecessors of the job, it may need to check the communications time to each machines.
4. **Overall Time Complexity:**
    - Now, to find the overall time complexity, we can sum up these complexities: `O(V + E) + O(Vlog V + E) + O(V * K + K * E)`.
    - Hence, the overall complexity: `O(Vlog V + V * K + K * E)`.

Further Improvements:

Since the problem is an NP-hard problem, we cannot find the optimal scheduling order in an efficient manner. However, there are still some possible improvements to the above implements:

- The execution order finding and job scheduling on machine steps are implemented separately. It may be merged into a single step such that the priority queue used during topological sorting may calculate the priority value based on the information about scheduled jobs on the machine.

- Due to the NP-hardness, the use of search-based and evolutionary approaches such as simulated annealing, genetic algorithms, and hill climing might be more viable. Due to the limited amount of time, implementing these algorithms integrating topological sorting became quite unfeasible.

References:

1. Kahn AB. Topological sorting of large networks. Communications of the ACM. 1962;5: 558–562.