

Parallel Optimal DAG Scheduling in Haskell

1 Introduction and Motivation

Multiprocessor schedules can be represented as a DAG-like structure, with edges representing dependencies between tasks. Finding an optimal parallel schedule is a known NP-hard problem.

2 Problem and Scope

We define our problem using standard scheduling notation $P, prec, c_{ij}, C_{max}$.

Formally, we have m identical processors, a directed acyclic graph (DAG) of tasks with precedence constraints, communication delays between processors, and the objective of minimizing total completion time (makespan).

For each task i , p_i, q_i, s_i , and $c_{i,j}$ represent the amount of time needed to execute the task, the processor assigned to the task, the scheduled starting time on the processor, and the communication delay between tasks i and j respectively. Note that each variable c_{ij} must be multiplied by an indicator function $\mathbf{1}[q_i \neq q_j]$, as the communication delay is 0 if two tasks are assigned to the same processor.

One important structural constraint on our DAG is that for an edge $i \rightarrow j$, the scheduled start time must be $s_j \geq s_i + p_i + c_{ij}\mathbf{1}[q_i \neq q_j]$.

Our goal is to prove that upon finding a “good-enough” feasible schedule (an upper bound), no schedule can do better, and therefore our found schedule is optimal.

3 Methodology

3.1 Sequential Algorithm (The Baseline)

Our baseline is DF/IHS (Depth-First / Implicit Heuristic Search), with CP/MISF (Critical Path / Most Immediate Successors First) as its heuristic, devised here. It uses a combination of branch-and-bound with HEFT-style scheduling.

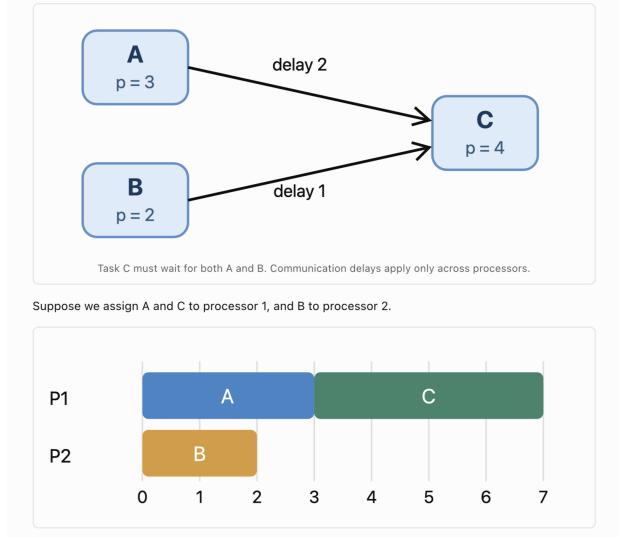


Figure 1: A simple sketch of the problem we are dealing with

As a depth-first branch-and-bound solver, it starts with a greedy schedule as an upper bound and explores alternative schedules to either prove the optimality of the greedy choice or update the choice.

Greedy list scheduling uses critical-path list scheduling, which calculates the longest computation-only path from the task to the end of the graph. This provides us with a good makespan upper bound, as a computation-only path represents a schedule in which each task is assigned to the same processor and the largest tasks are prioritized (minimizing delays of long dependency chains).

We must also keep track of processor availability, where the earliest possible start date for a given task t and processor q can be calculated as follows:

$ET(t, q) = \max(a_q, \max_{u \in \text{pred}(t)} [f(u) + c_{ut} \mathbf{1}[q_u \neq q_t]])$, where a_q is the current finish time on processor q . Given this time, we can branch on the partial schedule (the currently-ready processors and tasks - $(t, q) \in \text{Ready}(S) \times \{1..m\}$). Although the depth of this tree will be at most the number of tasks, this tree can grow in size exponentially, which is why the algorithm requires pruning.

The main idea of our pruning heuristic is to estimate a lower bound on the makespan of that branch (i.e the partial schedule explored). If it is the case that our partial schedule, S' , has $LB(S') \geq UB$, where UB is our initial greedy critical-path result. At this point we can stop our exploration of this branch.

Haskell's laziness is beneficial here - we check the partial schedule before constructing or evaluating the children, and if we can prune, we move on to the next branch. The laziness further allows us to keep fixed-size bounds, accumulations, and counters.

3.2 Parallel Algorithm (Our Plan)

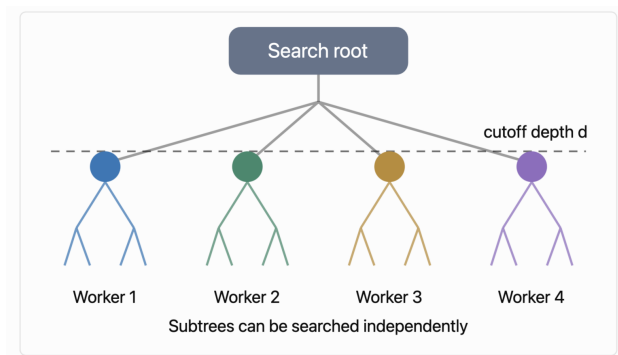


Figure 2: A high-level overview of our parallel implementation

In our parallel algorithm, after a certain cutoff depth d , we can spawn different workers for each subtree.

We make three design choices that affect the incumbent search, the best makespan we have found so far, and the search pruning methodology. The main change in comparison to the sequential algorithm is the method by which the subtrees learn about better incumbents found elsewhere.

We discuss these design choices in the sections below.

3.2.1 Design A

Design A represents independent parallelism where each worker in each subtree continues to search for incumbents. Pruning is weak here as other workers do not have any knowledge of the incumbents, therefore introducing redundant work through speculative computation.

Here we'll use `Pure Control.Parallel.Strategies (parBuffer + rdeepseq)`, where each subtree starts from the seed UB and returns its best leaf.

3.2.2 Design B

In Design B, we use a shared global incumbent that allows each worker to have knowledge of the best incumbent so far. This allows pruning to be much stronger compared to Design A.

Here we'll use `forkIO` to create lightweight Haskell threads, `IORef` to hold the shared incumbent, and `atomicModifyIORef'` for atomic updates.

Although B may outperform A by reducing redundant search, it has the tradeoff of synchronization and cache-coherence traffic which may impact performance.

3.2.3 Design C

Design C improves upon Design B by writing the shared incumbent state in discrete intervals rather than continuously. This introduces implementation complexity as each worker must be able to pause and record their DFS state in order to subsequently resume. Thus, we must retain the unfinished DFS stack or an explicit continuation rather than just calling a recursive function that runs to completion.

The fundamental tradeoff we explore in our candidate designs is that of getting the most recently calculated incumbent compared to the cost of coordination across workers.

4 Experimental Setup & Evaluation

We plan to test the following preliminary hypotheses and expect to add more to this list as we encounter different problem settings. Our input data will come from the same paper which devised the sequential algorithm, and we will attempt to find a GitHub repository with further cases.

Hypothesis 1: Parallelism helps more when the proof of optimality dominates, such as cases where the parity between the lower bound and the greedy upper bound solution is large.

Hypothesis 2: Shared incumbents (like Design B and Design C) improve pruning but increase overhead.

Hypothesis 3: Different DAG structures and the communication-to-computation ratio (CCR) affect search complexity.

We will measure the performance of the sequential branch-and-bound solver in comparison to the parallel solver with various numbers of threads (-N1, -N2, -N4, -N8).

We will compare:

- elapsed runtime,
- degree of pruning,
- number of nodes explored,
- speedup and parallel efficiency,
- effects of different initial frontier depths.

We will use GHC's runtime profiling tools (+RTS -s, spark profiles, event logs) to examine parallel behavior and resource consumption.