Task Parallelism

Comprehensive Review

DSC 208R — Parallel Data Processing and the Cloud

Contents

1 Motivation and Big Picture

Divide & Conquer. Task parallelism applies the classical divide-and-conquer strategy to data-engineering workloads: *place different tasks on different workers*; if they need the same dataset, simply *replicate* it to every worker:contentReference[oaicite:0]index=0.

This idea is attractive when:

- The task graph exposes substantial concurrency.
- The dataset is not too large to replicate across workers.
- Tasks are heterogeneous (e.g., feature engineering → model training → evaluation).

2 Core Concepts of Task Parallelism

2.1 Task Graphs and Topological Scheduling

Workloads are represented as Directed Acyclic Graphs (DAGs) of tasks. A scheduler performs a *topological sort* to respect dependencies when assigning tasks to workers:contentReference[oaicite:1]index=1.

2.2 Workers and Threads

A "worker" may be a whole node (process-level parallelism) or a single CPU core (thread-level parallelism). For example, Dask can treat a 4-node cluster with 4 cores each as 16 total workers:contentReference[oaicite:2]index=2.

2.3 Pros and Cons

- **Pros:** Simple mental model; independent workers imply low software complexity.:contentReference[oaicite:3]index=3
- **Cons:** Data replication wastes memory/storage; workers may sit idle when concurrency drops.:contentReference[oaicite:4]index=4

3 Degree of Parallelism

> *"The largest amount of concurrency possible in the task graph; i.e., how many tasks can run simultaneously."*:contentReference[oaicite:5]index=5

In the illustrative DAG (Figure 1), the degree of parallelism is 3; hence deploying more than three workers yields no additional speed-up.

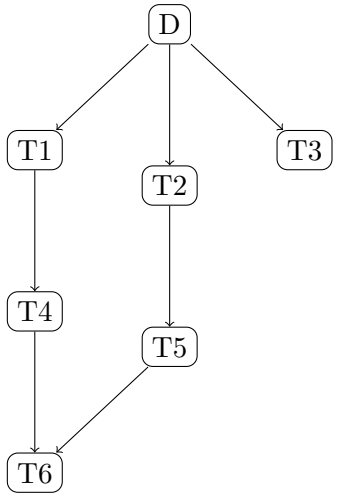

Figure 1: Example task graph with maximum concurrency of 3.

4 Mathematical Formulations

4.1 Speed-Up

$$\text{Speed-up}(n) = \frac{\text{Completion time with 1 worker}}{\text{Completion time with } n \text{ workers}} .$$

Ideal (linear) speed-up equals n. Real workloads often attain sub-linear speed-up due to limited degree of parallelism, data transfer, and scheduling overhead:contentReference[oaicite:6]index=6.

4.2 Lower Bound on Completion Time

The parallel completion time is lower-bounded by the *longest path* (critical path) in the task graph; idle time arises whenever the ready-task set is smaller than the worker pool:contentReference[oaicite:7]index=7.

5 Worked Example: Six-Task Workflow on Three Workers

Using the durations from the slides (in seconds):

$$\text{T1} = 10,\ \text{T2} = 5,\ \text{T3} = 15,\ \text{T4} = 5,\ \text{T5} = 20,\ \text{T6} = 10.$$

5.1 Serial vs. Parallel Runtime

- Serial runtime: 10+5+15+5+20+10 = 65 s.:contentReference[oaicite:8]index=8
- Parallel runtime (three workers): 35 s (see Gantt chart, Figure 2).:contentReference[oaicite:9]index=9

Hence speed-up = 65/35 ≈ 1.9, well below the ideal 3×.

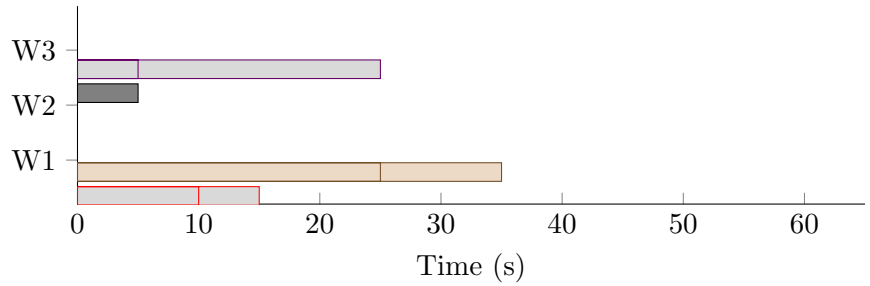

Figure 2: Illustrative Gantt chart: grey bars denote idle periods.

Why only 1.9×? Idle periods (grey) and the critical-path bound cap attainable speed-up. Adding more than three workers would *not* reduce runtime because the degree of parallelism is already saturated.

6 Algorithm Outline: Task-Parallel Scheduler

1. Perform a topological sort to identify ready tasks.
2. When a worker becomes free, assign it any ready task (FIFO, LIFO, or cost-aware heuristic).
3. Upon task completion, mark successors as ready; repeat until all tasks finish.
4. Optionally release idle workers early to save cloud costs when no further tasks can be assigned.:contentReference[oaicite:10]index=10

7 Practical Guidelines

- Match the number of workers to the workload's maximal degree of parallelism; more workers waste resources.

- Minimize replication overhead by keeping datasets small or by partitioning read-only reference data.
- Use frameworks like *Dask* to exploit thread-/process-level pools automatically (e.g., 4 nodes $\times$ 4 cores = 16 workers).:contentReference[oaicite:11]index=11
- Monitor idle time; consider elastic autoscaling to decommission idle workers and cut cloud costs.

8 Future Directions

- Integrate critical-path analysis with cost-based schedulers for cloud cost minimization.
- Explore hybrid task/data-parallel approaches to reduce replication while retaining independence.
- Improve predictive autoscaling policies that release idle workers without hurting completion time.

Conclusion

Task parallelism offers a simple yet powerful way to accelerate data-science pipelines by exploiting concurrency in task graphs. Its benefits, however, are bounded by degree of parallelism, data replication costs, and critical-path latency. Careful scheduling and resource sizing are essential to approach optimal performance.