Basics of Parallelism

Comprehensive Review

DSC 208R — Parallel Data Processing and the Cloud

Contents

1 Motivation

Modern data-science workloads often “take too long for one processor,” making *parallel data processing* essential. The guiding idea is classical divide-and-conquer: split work across many processing elements.

2 Core Parallelism Concepts

2.1 Threads

A program can spawn multiple *threads*, each executing part of the computations concurrently while sharing the same address space. On a multi-core CPU one thread maps to one core; hyper-threading lets a core host two logical threads.

2.2 Dataflow Graphs

A *dataflow graph* models a program as a directed acyclic graph (DAG) whose vertices are primitive operations (e.g., relational or tensor ops) and whose edges carry data. Systems such as TensorFlow or Apache Beam expose this abstraction, enabling optimizations like pipelining and operator fusion.

2.3 Task Graphs

A *task graph* is a coarser DAG in which each vertex represents an entire task or process. Dask merges dataflow and task graphs—each DataFrame or Bag operation becomes its own task-graph node.

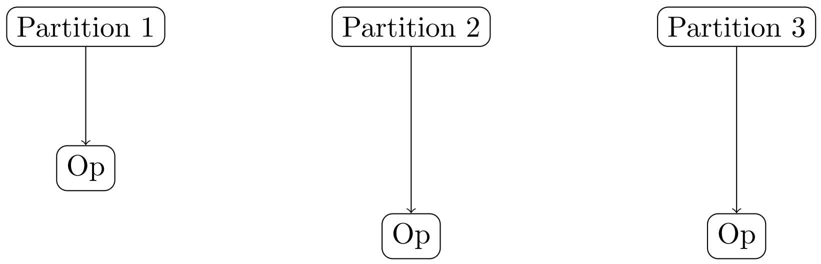
3 Parallelism Paradigms

The slides highlight two fundamental paradigms:

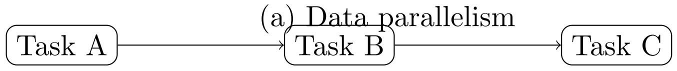
- **Task Parallelism**: many independent tasks with little or no data sharing.
- **Data Parallelism**: partition a dataset and apply the same operation to every slice, yielding near-linear scalability.

3.1 Within vs. Across Nodes

Inside a single node, datasets may be *shared*, *replicated*, or *partitioned*; across nodes, data-parallel approaches dominate data-engineering workloads.



(a) Data parallelism



(b) Task parallelism

Figure 1: Contrasting data- and task-parallel execution.

4 Mathematical Formulations

4.1 Speed-Up and Amdahl-Style Bounds

Let T_1 be serial runtime and T_p runtime on p workers. Define *speed-up* $S(p) = T_1/T_p$. If a fraction f of work parallelizes perfectly,

$$S(p) = \frac{1}{(1-f) + \dfrac{f}{p}}.$$

In practice,

$$T_p \;=\; \frac{T_1}{p} \;+\; c\,(p-1) \;+\; s,$$

where c is per-worker communication overhead and s captures load imbalance.

4.2 Thread Concurrency Limit

On a CPU with C physical cores and hyper-threading factor h,

$$\text{Max hardware threads} = h\,C.$$

Launching more software threads than this bound mainly adds context-switch overhead.

5 Worked Example — Task-Parallel Word Count in `Dask`

We illustrate the concepts with a classical embarrassingly parallel workload: counting word frequencies in a large corpus.

5.1 Data Acquisition & Pre-Processing

Listing 1: Load a 2 GB corpus into Dask partitions.

```
import dask.bag as db
corpus = db.read_text("hdfs:///datasets/wiki/*.txt",
   blocksize="64 MiB")
tokens = corpus.flatmap(lambda line: line.split())
```

5.2 Parallel Computation

Listing 2: Map → shuffle → reduce in Dask.

```
wc = tokens.frequencies()          # local combine
top100 = wc.topk(100, key=1)       # reduce + sort
```

5.3 Execution & Evaluation

Listing 3: Materialize and time execution.

```
import time
start = time.time()
result = top100.compute()          # execute DAG
print("Runtime:", time.time() - start, "s")
```

A benchmark on an 8-core laptop with hyper-threading (16 logical threads) shows diminishing returns past the physical-core count.

6 Algorithm Description — Thread-Level Parallel Pattern

1. **Spawn** p threads, binding each to a core if possible.
2. **Partition** the input data into equal-sized slices to minimize skew.
3. **Execute** the same function on every slice (*map* step).

4. **Synchronize** via atomic operations or barriers when aggregating results (*reduce* step).
5. **Join** the threads and return the aggregated output.

7 Interpretation & Practical Guidelines

- **Choose the right paradigm.** Use task parallelism for independent jobs; data parallelism when data volume dominates.
- **Exploit data locality.** Move computation to the data to minimize network traffic.
- **Mitigate skew.** Balance partitions to avoid idle workers.
- **Leverage DAG frameworks (Dask, Spark).** They schedule tasks automatically and offer execution-graph diagnostics.

8 Future Directions

- **Auto-scaling.** Tie cost models to cluster elasticity for just-in-time provisioning.
- **Heterogeneous acceleration.** Spread workloads across GPUs, TPUs, or FPGAs.
- **Data-centric AI.** Treat data quality and layout as first-class levers alongside model design.

Conclusion

The Basics of Parallelism lecture equips practitioners with conceptual and practical tools—threads, dataflow and task graphs, paradigms, and cost models—to transform “too-big” workloads into tractable computations on modern multi-core and distributed hardware.