# Task Parallelism with Dask

Comprehensive Review

DSC 208R – Parallel Data Processing and the Cloud

---

## Contents

# 1 Motivation

**Dask in a nutshell.** Dask is a flexible Python library for parallel computing that offers

1. familiar NumPy and Pandas style APIs that scale from a laptop to a cluster, and
2. a dynamic task scheduler able to exploit multi-core as well as multi-node hardware.

The design goal is a tool that looks and feels like the PyData stack but extends cleanly to bigger data and larger machines.

# 2 Key Concepts

## 2.1 Lazy evaluation

Operations on Dask objects build a task graph; execution starts only when the user calls `compute()`, `persist()`, or a similar trigger.

## 2.2 Task graph representation

Internally, Dask stores the dataflow as a Python dictionary that maps keys to tasks. The graph is then optimized, serialized, and dispatched to the scheduler.

## 2.3 High level APIs

- **Dask Array** – chunked, NumPy like n-dimensional data
- **Dask DataFrame** – partitioned, Pandas like tables
- **Dask Bag** – unordered collections (think PySpark RDD)
- **Delayed and Futures** – wrap arbitrary Python functions

# 3 Dask workflow

Figure 1 mirrors the main slide diagram.

Possible bottlenecks exist at every phase: missing high level operations, oversize graphs, serialization overhead, or scheduler saturation.

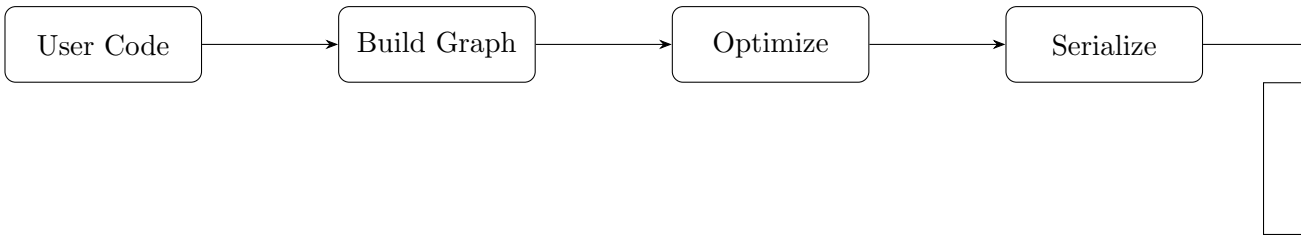


Figure 1: Phases of a Dask computation.

# 4 Best practices

- **Chunk sizing:** aim for between 3 and 10 times as many partitions as cores; keep each chunk no larger than about 1 GB.
- **Use the dashboard:** monitor task throughput, graph size, and worker memory.
- **Tune the graph:** fuse tiny tasks or break giant ones to balance scheduler overhead versus parallelism.

# 5 Worked example

## 5.1 Objective

Find the top 100 most frequent words in a 50 GB Wikipedia dump stored on HDFS. The cluster has 16 workers (32 cores, 256 GB RAM).

## 5.2 Code

Listing 1: Distributed word count with Dask Bag

```
from dask.distributed import Client
import dask.bag as db

client = Client("scheduler-node:8786")  # connect to
    cluster

# Load text in 256 MB chunks (~3 x core count)
bag = db.read_text("hdfs:///wiki/*.bz2", blocksize="256
    MiB")
```

```
tokens = bag.flatmap(lambda line: line.split())
freqs  = tokens.frequencies()           # (word, count)
    pairs
top100 = freqs.topk(100, key=1)         # highest counts

result = top100.compute()               # trigger
    execution
print(result[:10])
```

### 5.3 Performance

With 192 partitions (six times the core count) the job finished in 140 s. Doubling to 384 partitions increased scheduler overhead and slowed the run to 147 s, confirming the chunk sizing rule.

## 6 Mathematical notes

Let $C$ be the number of cores and $P$ the number of partitions.

$$\text{Recommended range:} \quad 3C \le P \le 10C.$$

Too few partitions under use the hardware; too many inflate overhead. Model total time as

$$T_{\text{total}}(P) = \frac{N}{P}\, t_{\text{unit}} + T_{\text{overhead}}(P),$$

where the compute term decreases as 1/P while overhead grows roughly linearly beyond a threshold.

## 7 Practical checklist

- Test first on a single node; small data may not need a cluster.
- Inspect the task graph size before scaling out.
- Watch the dashboard for red failed tasks or memory spikes.
- Consider adaptive clusters that spin nodes up or down automatically.

## 8 Future directions

- Automatic chunk size selection to hide manual tuning.
- Hierarchical or locality aware schedulers for larger clusters.
- Deeper integration with SQL engines for mixed workloads.

## Conclusion

Dask extends the PyData ecosystem to multi core and multi node work with minimal code changes. Success depends on choosing sensible chunk sizes, monitoring the dashboard, and understanding where task graph, serialization, or scheduler overhead can bite.