

UNIT 4

MapReduce

(8 Hours) — Cloud Computing (CACS402)

Disclaimer: These notes were prepared as a study aid and may contain errors or omissions. Please cross-check important definitions, figures and product details (especially vendor services such as Google App Engine, Microsoft Azure and Aneka, which change often) against your prescribed textbook, class notes and official documentation before relying on them in an exam.

4.1 Introduction to Parallel Computing

Parallel computing solves a large problem by dividing it into smaller parts that are executed simultaneously on multiple processors or machines. It is essential for processing very large data sets (“big data”) that a single computer cannot handle in reasonable time.

Type	Description
Bit-level / Instruction-level	Parallelism inside a single processor (wider words, pipelining).
Data parallelism	Same operation applied to different pieces of data at the same time (MapReduce follows this).
Task parallelism	Different tasks run concurrently on the same or different data.
Shared-memory systems	Processors share one memory space (multi-core, SMP).
Distributed-memory systems	Each node has its own memory; nodes communicate by messages (clusters, clouds).

Speedup and efficiency

- **Speedup** $S = T_1 / T_p$, where T_1 is time on one processor and T_p is time on p processors.
- **Efficiency** $E = S / p$.
- **Amdahl's law:** if a fraction f of the program is sequential, the maximum speedup on p processors is $S = 1 / (f + (1 - f)/p)$.

Example 4.1 — Amdahl's law

A job has 10% sequential work ($f = 0.1$). Find the speedup on 10 processors and the limit with unlimited processors.

Solution

$$S = 1 / (0.1 + 0.9/10) = 1 / (0.1 + 0.09) = 1 / 0.19 \approx \mathbf{5.26}.$$

$$\text{Efficiency} = 5.26 / 10 \approx 52.6\%.$$

As $p \rightarrow \infty$, $S \rightarrow 1 / f = \mathbf{10}$. The sequential part limits speedup.

Challenges of parallel programming: task decomposition, synchronisation, communication overhead, load balancing, fault tolerance and debugging. MapReduce hides most of these from the programmer.

4.2 Map-Reduce Model

MapReduce is a programming model and framework, introduced by Google (Dean and Ghemawat, 2004), for processing large data sets in parallel on clusters of commodity machines. The programmer writes only two functions; the framework handles partitioning, scheduling, communication and failures.

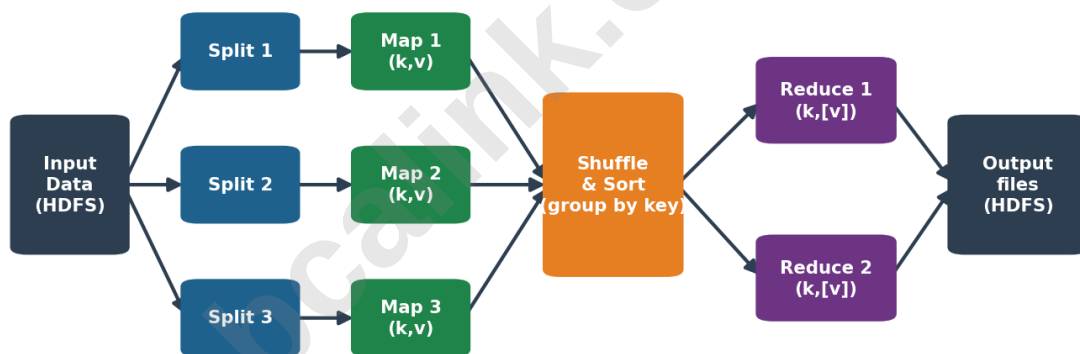


Figure 4.1 — MapReduce data flow

The two functions

- **Map:** $(k_1, v_1) \rightarrow \text{list}(k_2, v_2)$. Processes each input record and emits intermediate key-value pairs.
- **Reduce:** $(k_2, \text{list}(v_2)) \rightarrow \text{list}(k_3, v_3)$. Receives all values for one key and combines them into the result.

Execution phases

Phase	Description
Input splitting	Input files (in a distributed file system such as GFS/HDFS) are divided into splits, typically 64–128 MB blocks.
Map	Each split is processed by a map task, preferably on the node that stores the data (data locality).
Combine (optional)	A local mini-reducer merges pairs on the map node to reduce network traffic.
Shuffle and sort	Intermediate pairs are partitioned by key, moved to reducers and sorted so that equal keys are grouped.
Reduce	Each reducer processes each key group and writes output to the file system.

Example 4.2 — Word count

Input lines: “cloud is big”, “big data in cloud”. Show the MapReduce word count.

Solution

Map output: (cloud,1) (is,1) (big,1) | (big,1) (data,1) (in,1) (cloud,1).

Shuffle/sort groups by key: big → [1,1]; cloud → [1,1]; data → [1]; in → [1]; is → [1].

Reduce (sum): (big,2) (cloud,2) (data,1) (in,1) (is,1).

Pseudo-code for word count:

Pseudo-code

```
map(String key, String value): // value = a line of text
```

```
    for each word w in value: EmitIntermediate(w, 1)
```

```
reduce(String word, Iterator counts):
```

```
    sum = 0; for each c in counts: sum += c
```

```
    Emit(word, sum)
```

Fault tolerance

- The master pings workers periodically; a failed worker’s tasks are re-executed on other nodes.
- Slow tasks (“stragglers”) are handled by speculative (backup) execution.
- Map and reduce functions must be deterministic so re-execution gives the same output.

4.3 Applications of MapReduce

Application	How MapReduce is used
Word count / text processing	Count word frequencies, build inverted indexes for search engines.
Log analysis	Map extracts fields from web-server logs, reduce counts hits per URL or user.
Distributed grep and sort	Map emits matching lines; the framework's sort is used for large-scale sorting.
Web-link graph analysis	Compute reverse web-link graph and PageRank iterations.
Machine learning and data mining	Clustering (k-means), classification, recommendation systems.
Data warehousing / ETL	Hive and Pig compile queries to MapReduce jobs.
Bioinformatics and scientific computing	Genome sequence alignment, simulation post-processing.
Advertising and social media analytics	Click-stream analysis, trend detection.

Example 4.3 — Inverted index

Map: for each document ID d and each word w , emit (w, d) .

Reduce: for each word w , output $(w, \text{sorted list of document IDs})$. This index lets a search engine find documents containing a word quickly.

4.4 Parallel Efficiency of Map-Reduce

The performance of a MapReduce job depends on how well the work is spread and how much overhead the framework adds.

- **Speedup** = T_1 / T_n and **parallel efficiency** = Speedup / n , where n is the number of nodes.
- Ideal efficiency is 1 (100%); real jobs are lower due to overheads.

Factors affecting efficiency

Factor	Effect / remedy
Data locality	Running map tasks where data resides avoids network transfer; the scheduler tries node-local, then rack-local.
Shuffle cost	Moving intermediate data is the main overhead; use combiners and compression to reduce it.
Load imbalance / data skew	A few keys with huge value lists overload some reducers; use custom partitioners.
Stragglers	Slow nodes delay job completion; speculative execution helps.
Task start-up overhead	Too many small tasks waste time; choose suitable split size and number of reducers.
Sequential phases	Reduce cannot start finishing until all maps complete (Amdahl's law).

Example 4.4 — Parallel efficiency

A job takes 800 minutes on 1 node and 25 minutes on 40 nodes. Compute speedup and efficiency.

Solution

Speedup $S = 800 / 25 = 32$.

Efficiency $E = 32 / 40 = 0.8 = 80\%$.

The missing 20% is due to shuffle, start-up, skew and other overhead.

4.5 MapReduce Infrastructure

A MapReduce system consists of a distributed file system plus a job-execution layer. The best-known open-source implementation is **Apache Hadoop**.

Components

Component	Role
HDFS (Hadoop Distributed File System)	Stores files as large blocks (default 128 MB in Hadoop 2+), replicated (default 3×) across DataNodes. NameNode holds metadata.
JobTracker / TaskTracker (Hadoop 1)	JobTracker schedules jobs and monitors TaskTrackers, which run map and reduce tasks on each node.
YARN (Hadoop 2+)	ResourceManager allocates cluster resources; NodeManagers run containers; an ApplicationMaster manages each job.
Master node	Coordinates the job, assigns tasks, tracks progress and failures.
Worker nodes	Execute map and reduce tasks and store data blocks.

Job execution steps

- Client submits the job (code, configuration, input/output paths).
- Framework computes input splits and the master schedules map tasks near the data.
- Map outputs are written to local disk, partitioned by key.
- Reducers fetch (shuffle), merge and sort their partitions, then run the reduce function.
- Final output is written to HDFS; job status is reported to the client.

Related ecosystem

- **Hive** — SQL-like queries; **Pig** — data-flow scripting; **HBase** — column-oriented NoSQL store.
- **Spark** — in-memory processing, faster for iterative algorithms.
- Cloud offerings: Amazon EMR, Google Dataproc, Azure HDInsight run Hadoop/Spark as managed services.

Note: Google's original MapReduce ran on GFS (Google File System); Hadoop reproduces this design using HDFS.

Key Takeaways — Unit 4

- ✓ Parallel computing splits work across processors; speedup = T_1/T_p , efficiency = S/p , limited by Amdahl's law.
- ✓ MapReduce: Map turns input into (key, value) pairs; the framework shuffles and sorts by key; Reduce aggregates each key.
- ✓ Fault tolerance comes from re-execution of failed tasks and speculative execution of stragglers.
- ✓ Applications include word count, log analysis, indexing, machine learning and ETL.
- ✓ Efficiency depends on data locality, shuffle cost, skew and stragglers.
- ✓ Infrastructure: HDFS for storage, master/worker (JobTracker/YARN) for execution; Hadoop is the standard implementation.

— *End of Unit 4* —

bcalink.com