MapReduce Implementation in Go

Alex Shin
Yale University

Mati Hassan
Yale University

Abstract

We present a distributed implementation of MapReduce based on Google's seminal paper, focusing on the core execution and fault tolerance mechanisms. Our system implements the fundamental MapReduce architecture using Go, featuring a master-worker model with RPC-based communication. The implementation supports parallel task execution, handles worker failures through a timeout mechanism, and ensures atomic output operations. We validate our implementation through a comprehensive test suite including word count, distributed grep, and indexing applications. While we intentionally omit certain optimizations like data locality (section 3.4) to maintain simplicity, our system successfully demonstrates the key principles of distributed data processing using the MapReduce programming model. Results show that the implementation achieves reliable parallel execution and fault tolerance while maintaining the simplicity of the original MapReduce abstraction.

1 Introduction

MapReduce has revolutionized large-scale data processing by providing a simple programming model that abstracts away the complexities of distributed computation. Building on MIT's 6.5840 Lab 1 skeleton code, we have developed a distributed MapReduce implementation that closely follows the architecture described in Google's original MapReduce paper by Dean and Ghemawat.

Our implementation focuses on three core components: a master node that coordinates execution, multiple worker nodes that perform computations, and a fault-tolerant communication system using RPC. The master node manages task distribution and monitors worker health, while workers execute map and reduce operations on assigned data chunks. We implement key features including:

- Distributed task execution with parallel processing capabilities
- Fault tolerance through worker failure detection and task reassignment
- Atomic output handling to ensure consistency
- A flexible plugin architecture supporting various applications

While our implementation omits certain optimizations like data locality to maintain design simplicity, it provides a complete demonstration of MapReduce's fundamental principles. The system supports both basic operations like word counting and more complex applications such as distributed grep and indexing, validating its practical utility for distributed data processing tasks.

The remainder of this paper describes our implementation in detail. Section 2 outlines the core architecture and execution workflow. Section 3 discusses our fault tolerance mechanisms and task management approach. Section 4 presents our testing methodology and validation results. We conclude with a comparison to related work and potential future improvements.

2 Core Architecture

The framework operates through three main stages: Map, Shuffle, and Reduce.

2.1 Master Node

- Distributes Map and Reduce tasks to available Worker Nodes.
- Tracks task statuses (IDLE, IN_PROGRESS, COMPLETED) and monitors worker states.
- Implements fault tolerance by detecting task timeouts (10 seconds) and reassigning uncompleted tasks to other workers.

- Manages task scheduling and facilitates the shuffle stage, ensuring intermediate data is correctly partitioned and delivered to Reduce workers.

2.2 Worker Nodes

- Execute Map tasks to process input files and produce intermediate key-value pairs.
- Write intermediate results to local files partitioned by hash values.
- Execute Reduce tasks by aggregating intermediate key-value pairs, applying the reduce function, and producing the final output.

2.3 Workflow

1. The Master node partitions input data into chunks and assigns Map tasks to Worker nodes.
2. Workers process Map tasks, writing intermediate results to files (e.g., `mr-X-Y`, where X is the map task index and Y is the Reduce task ID).
3. Upon completing all Map tasks, the Master assigns Reduce tasks, providing workers with locations of intermediate files.
4. Reduce workers fetch intermediate data, sort it by key, and apply the reduce function to generate outputs (e.g., `mr-out-Z`, where Z is the reduce task index).
5. The Master monitors task completions and signals workers to exit once all tasks are done.

2.4 Execution Overview (Section 3.1)

Following Figure 1 in the paper, our implementation includes:

1. Master node that coordinates execution
2. Worker nodes that perform map and reduce operations
3. Task assignment and data flow matching the paper's design

The key execution steps mirror those outlined in Section 3.1:

```
// Master initialization (step 1)
func MakeMaster(files []string, nReduce int) *Master {
    c := Master{}
    c.files = files
    c.nReduce = nReduce
    c.mapTasks = make([]MapTask, len(files))
    c.reduceTasks = make([]ReduceTask, nReduce)
    // ...
}
```

2.5 Master Data Structures (Section 3.2)

Our implementation closely follows the master state management described in Section 3.2:

```
type Master struct {
    files    []string
    nReduce  int
    mapTasks    []MapTask
    reduceTasks []ReduceTask
    mapTasksRemaining    int
    reduceTasksRemaining int
    mu sync.Mutex
}
```

This tracks:

- Task states (idle, in-progress, completed)
- Worker assignments
- Intermediate file locations

Just as described in the paper's master data structures section.

2.6 Fault Tolerance (Section 3.3)

We implemented the fault tolerance mechanisms described in Section 3.3:

2.6.1 Worker Failure Handling

```
const TASK_TIMEOUT = 10 * time.Second

func (c *Master) findAvailableMapTask(workerId string) *MapTask {
   // Check for timed out tasks as described in Section 3.3
   for i := range c.mapTasks {
      if c.mapTasks[i].Status == IN_PROGRESS &&
        time.Since(c.mapTasks[i].StartedAt) > TASK_TIMEOUT {
            c.mapTasks[i].Status = IDLE
      }
   }
    // ...
}
```

This implements the paper's worker failure detection and recovery mechanism, though we use a simpler timeout-based approach rather than the ping system described in the paper.

2.6.2 Atomic Operations

Following Section 3.3's "Semantics in the Presence of Failures", we implement atomic output handling:

```
func (w *worker) doMap(task *MapTask) error {
    // Write to temporary files first
    tempFile, err := ioutil.TempFile("", "map-")
    // ...
    // Atomic rename on completion
    finalName := fmt.Sprintf("mr-%d-%d", task.Task.Index, i)
    if err := os.Rename(f.Name(), finalName); err != nil {
        return fmt.Errorf("cannot rename temp file: %v", err)
    }
}
```

2.7 Deviations from Paper

Our implementation differs from the paper in several ways:

1. **Locality (Section 3.4)**
    - We did not implement the locality optimizations
    - All file access is local/network agnostic
    - This would be a key optimization for a production system
2. **Task Granularity (Section 3.5)**
    - While we support configurable M (map) and R (reduce) values, we don't implement the dynamic sizing described in the paper
    - Our implementation uses simpler fixed task sizes
3. **Backup Tasks (Section 3.6)**
    - We use timeouts instead of the backup task mechanism
    - This is simpler but potentially less efficient for stragglers

3 Design Choices & Tradeoffs

3.1 Task Assignment Protocol

Unlike the paper's push model, we implemented a pull-based task assignment:

```
func (w *worker) run() {
    for {
        task, err := w.requestTask()
        // ...
    }
}
```

3.2 Tradeoffs:

Advantages:

- Simpler worker failure handling
- Natural load balancing

Disadvantages:

- More network traffic from polling
- Doesn't benefit from locality optimizations

3.3 Intermediate Data Management

Following Section 3.1, we implement intermediate file management:

```
// Map output
intermediateFiles := make([]*os.File, task.NReduce)
for i := 0; i < task.NReduce; i++ {
    tempFile, err := ioutil.TempFile("", "map-")
    // ...
}

// Reduce input
for _, filename := range task.Locations {
    file, err := os.Open(filename)
    // ...
}
```

This matches the paper's approach but without the GFS integration.

4 Testing & Validation

Our testing suite ensures compliance with the key guarantees outlined in Section 3.3 of the MapReduce paper:

- Deterministic output for deterministic functions
- Fault tolerance and recovery
- Efficient parallel execution
- Atomic output handling

In addition to three production-style applications – `wc.go`, `grep.go`, and `indexer.go` – we have six testing applications to validate our system:

- **mtiming.go** (for map parallelism)
- **rtiming.go** (for reduce parallelism)
- **jobcount.go** (to verify task assignment)
- **crash.go** (to test fault tolerance under failure conditions)
- **nocrash.go** (baseline test for fault tolerance without failures)
- **early_exit.go** (to check worker behavior during early task completion)

5 Related Work

5.1 Comparison to Other Implementations

- **Hadoop [2]**: An open-source Java-based MapReduce implementation. Unlike Hadoop, our Go-based implementation avoids HDFS, focusing on local storage and core MapReduce functionalities.

- **Apache Spark [3]**: Employs in-memory Resilient Distributed Datasets (RDDs) to minimize I/O costs, outperforming traditional MapReduce for iterative workloads. Our project remains closer to the original MapReduce model to highlight foundational concepts.

5.2 Integration of Distributed Grep

Inspired by Section 2.3 of the MapReduce paper, we implemented Distributed Grep to search for patterns between distributed input files. This application validates the flexibility of the framework and its suitability for real-world distributed data processing.

5.3 Locality Optimization

Dean et al. [1] emphasize data locality optimization to reduce network overhead by scheduling tasks on nodes where data reside. Although omitted in this project, future iterations could incorporate this feature by simulating a shared filesystem or implementing data placement strategies.

6 Conclusion

Our implementation successfully demonstrates the core MapReduce architecture described in Section 3 of the paper. Although we omitted some optimizations (notably locality awareness), the system provides the fundamental guarantees of:

- Fault tolerance
- Parallel execution
- Atomic outputs
- Simple programming model

The implementation serves as a practical demonstration of the main concepts of the paper.

7 References

1. Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified data processing on large clusters. Retrieved from https://research.google.com/archive/mapreduce-osdi04.pdf
2. Apache Hadoop. (n.d.). Retrieved from https://hadoop.apache.org/
3. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., ... & Stoica, I. (2012). Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In 9th USENIX symposium on networked systems design and implementation (NSDI 12) (pp. 15-28). Retrieved from https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf