MapReduce Implementation in Go

Overview

MapReduce is a programming model and implementation for processing large datasets in a distributed manner, originally developed by Google.

We utilized the MIT skeleton code from their 6.5840 Lab 1: MapReduce, which featured a sequential MapReduce implementation for comparison. Our goal was to extend this work by implementing a distributed version based on the foundational principles outlined in the Google MapReduce paper. Specifically, we followed Section 3 ("Implementation") of Google's seminal MapReduce paper by Dean and Ghemawat [1], focusing on core execution and fault tolerance mechanisms. However, we intentionally omitted the data locality optimization (Section 3.4) to maintain simplicity in our design.

Our system architecture consists of a Master Node coordinating multiple Worker Nodes, with communication handled through gRPC for efficient task distribution and progress tracking. To further validate our implementation, we developed a MapReduce application called Distributed Grep, inspired by Section 2.3 of the Google MapReduce paper. This allowed us to test and demonstrate the effectiveness of our distributed MapReduce system.

Core Architecture

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

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.

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.

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.

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)
```

```
    // ...
}
```

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.

Fault Tolerance (Section 3.3)

We implemented the fault tolerance mechanisms described in Section 3.3:

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.

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)
    }
}
```

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

Design Choices & Tradeoffs

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()
        // ...
    }
}
```

Tradeoffs:

- Advantages:
  - Simpler worker failure handling
  - Natural load balancing
- Disadvantages:
  - More network traffic from polling
  - Doesn't benefit from locality optimizations

2. 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.

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)

Related Work

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.

Integration of Distributed Grep:

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

Locality Optimization:

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

Conclusion

Our implementation successfully demonstrates the core MapReduce architecture described in Section 3 of the paper. While 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 paper's core concepts.

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