Practical Work 4: Word Count using MapReduce Paradigm

Group ID: 18

1 Implementation Design Choice

1.1 Problem Statement

The objective of this practical work is to implement a Word Count application based on the MapReduce paradigm. Students are encouraged to simulate the distributed MapReduce architecture using available tools and frameworks.

1.2 Proposed Solution: Multi-threading in Python

We decided to simulate the distributed MapReduce architecture on a single machine using **Threading** module in the Python language.

Rationale:

- **Efficiency:** Creating a full distributed system over a network introduces significant latency and complexity regarding connection management. Threads share the same memory space, allowing for faster data exchange and lower overhead.
- **Simplicity:** Python provides a clean and readable syntax while still allowing explicit synchronization between "nodes" (threads) using Locks, which maintains understanding of concurrent system challenges.
- **Parallelism:** This approach satisfies the core requirement of parallel processing. Each thread acts as an independent worker node (Mapper) processing a specific data chunk simultaneously.
- **Data Structures:** Python's built-in dictionary and collections module (e.g., `defaultdict`) provide efficient key-value storage, simplifying the aggregation process compared to manual array management.

2 System Architecture

Our implementation adapts the MapReduce flow for a shared-memory environment.

2.1 Workflow Description

1. **Input Reading:** The main program reads the entire content from the input file (`input.txt`) and stores it in memory for processing.
2. **Data Splitting:** The text is tokenized into individual words using Python's `split()` method. These words are then divided into roughly equal chunks based on the number of mapper threads (`NUM_MAPPERS`). The last chunk receives any remaining words to ensure complete coverage.
3. **Mapping Phase (Parallel Processing):** Each mapper thread:
    - Receives a specific chunk of words to process

- Counts word occurrences locally using `defaultdict(int)`
- Maintains thread-local storage to avoid premature synchronization overhead

4. **Reducing Phase (Aggregation):** After local counting, each thread:
   - Acquires a mutual exclusion lock (`threading.Lock`)
   - Merges its local counts into the global result dictionary
   - Releases the lock in a `try-finally` block to guarantee unlocking even if exceptions occur
5. **Synchronization:** The main thread uses `thread.join()` to wait for all mapper threads to complete before displaying results.
6. **Output:** Final results are sorted alphabetically and printed to the console, showing each unique word with its total count.

2.2 Architecture Diagram

Figure 1 illustrates the data flow from the input file through the parallel threads to the final aggregation.

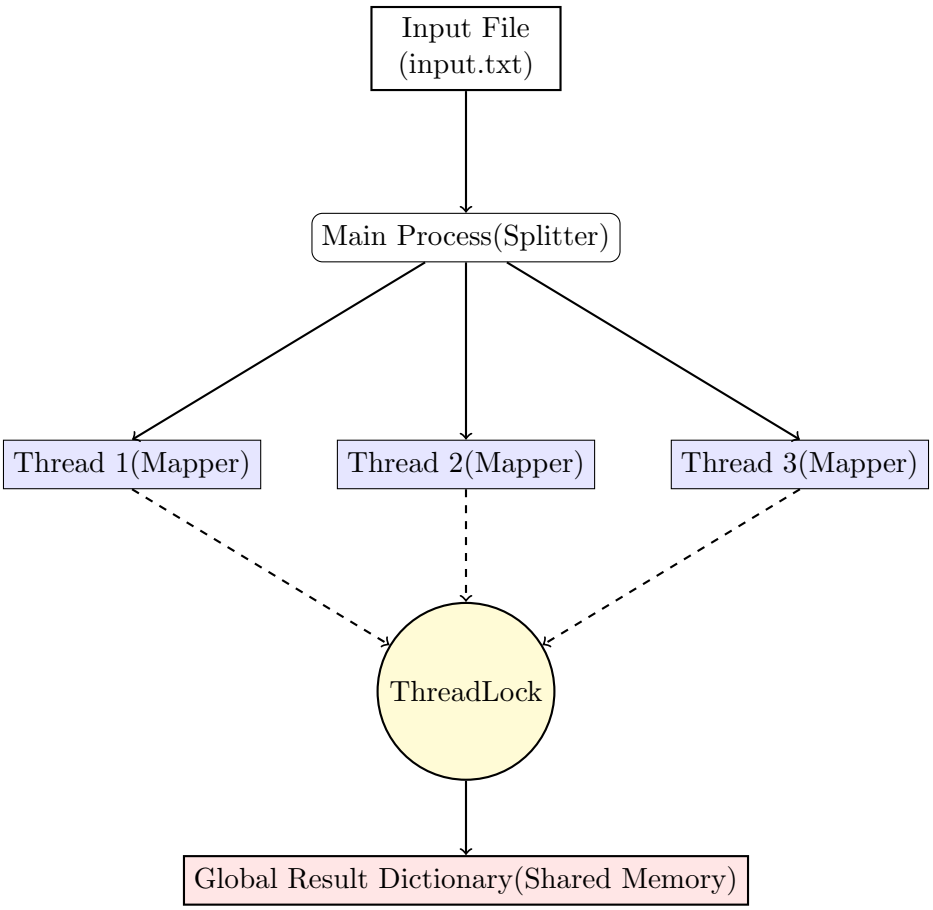

Figure 1: Architecture of the Multi-threaded MapReduce System

3 Implementation Details

3.1 Key Components

Global Variables:

- `NUM_MAPPERS`: Configurable number of parallel threads (default: 3)

- `global_results`: Shared dictionary storing final word counts
- `lock`: `threading.Lock()` object for mutual exclusion

3.2 Mapper Function

Below is the core logic for the Mapper function with detailed inline comments:

```
def mapper(args):
    thread_id = args['id']
    text_chunk = args['chunk']

    # STEP 1: Local word counting (thread-safe by design)
    local_counts = defaultdict(int)
    words = text_chunk.split()  # Tokenize by whitespace

    for word in words:
        local_counts[word] += 1  # Increment count

    print(f"[Mapper {thread_id}] Found {len(local_counts)} unique words.")

    # STEP 2: Merge with global results (CRITICAL SECTION)
    global global_results, global_count

    lock.acquire()  # Enter critical section
    try:
        for word, count in local_counts.items():
            if word in global_results:
                global_results[word] += count
            else:
                global_results[word] = count
                global_count += 1
    finally:
        lock.release()  # Always release lock
```

Listing 1: The Mapper Thread Function with Comments

3.3 Thread Safety Considerations

The **lock mechanism** is critical for correctness:

- Without locking, multiple threads could read-modify-write the same dictionary entry simultaneously, causing **race conditions** and incorrect counts.
- The `try-finally` pattern ensures the lock is always released, preventing **deadlocks**.
- Local counting minimizes time spent in the critical section, improving **parallelism efficiency**.

4 Results and Testing

4.1 Test Environment

- **Platform:** Python 3.x
- **Configuration:** 3 mapper threads (configurable)
- **Input:** `input.txt` containing sample text

4.2 Verification

The implementation was tested with various input files to ensure:

1. **Correctness:** Word counts match sequential counting algorithms
2. **Thread Safety:** No race conditions occur during concurrent updates
3. **Completeness:** All words are counted exactly once
4. **Scalability:** Performance improves with additional threads (up to hardware limits)

4.3 Sample Output

The program produces sorted output in the format:

```
--- FINAL RESULTS ---
apple: 3
banana: 2
cherry: 5
...
```

5 Task Division

The workload was distributed among group members to ensure parallel development:

- Defined the data structures (dictionary and collections) and overall project organization.
- Implemented the Mapper logic, specifically focusing on word splitting and local counting using `defaultdict`.
- Developed the Reducer logic (aggregation) and implemented Lock synchronization for thread safety.
- Managed file Input/Output operations and the logic for splitting the input file by words (word-boundary splitting).
- Conducted testing, debugging, and verified the accuracy of the word counts.
- Authored this LaTeX report and designed the system architecture diagram.

6 Conclusion

This practical work successfully demonstrates the MapReduce paradigm using Python's threading capabilities. Key achievements include:

- **Parallel Processing:** Multiple threads process data chunks simultaneously, simulating distributed mapper nodes.
- **Thread Safety:** Proper synchronization using locks ensures data consistency without race conditions.
- **Scalability:** The configurable `NUM_MAPPERS` parameter allows easy adjustment to match system resources.
- **Simplicity:** Python's high-level abstractions (`defaultdict`, `threading`) make the implementation concise and maintainable.

While this single-machine simulation doesn't capture network communication overhead present in true distributed systems, it effectively illustrates the core MapReduce concepts: data partitioning, parallel mapping, and result aggregation. The implementation provides a solid foundation for understanding distributed computing principles.

6.1 Future Improvements

- Implement true distributed processing using frameworks like Apache Hadoop or Apache Spark
- Add support for custom word tokenization (handling punctuation, case-insensitivity)
- Implement performance benchmarking with varying thread counts
- Add visualization of thread execution timeline