1. Introduction

Text word counting is a classical problem in text processing, frequently applied in data analysis, natural language processing, and search engines. The problem of sequential counting of words in growing larger text files becomes slower and inefficient. In order to speed up the word-counting process on multi-core processors by dividing the workload among multiple threads, this project uses OpenMP in C++ to implement a Parallel Word Count system.

---

2. Project Objectives

1. Split a big text file into separate segments.
2. Handle each segment at the same time, getting as well as counting words.
3. Combine the outputs from all threads in an efficient way.
4. Track performance (tokens number, time execution, top-N words).
5. Evaluate sequential against parallel execution (optional extension).

---

3. Methodology

3.1 Input Reading

The program accepts:

- `input.txt` → the file to analyze
- `topN` → (optional) number of top frequent words to show

It first loads the entire file into a vector of lines.

3.2 Parallel Processing

Each thread receives a portion of the lines using:

```
#pragma omp for schedule(static)
```

- Every thread maintains a separate local hashmap (unordered_map) which helps in preventing false sharing and also in avoiding the contention for locks.
- 3.3 Tokenization
- A character is included in the word count if it is alphanumeric:
- isalnum(c)
- Individually, each token is turned to lowercase and placed in the local map.
- 3.4 Merging Phase
- Once all threads are done, the local maps are combined into one global frequency map.
- Simultaneously, the total token count is gathered.
- 3.5 Producing Final Output
- The program displays the following information:
- Execution mode
- Number of threads
- Total tokens
- Number of unique words
- Execution time in milliseconds
- Top N most frequent words

---

4. Code Structure Overview

Sections

1. **Reading of Files**
2. **Tokenization in Parallel**
3. **Local Map Building**
4. **Global Map Reduction**
5. **Finding Top-N Words**
6. **Performance Evaluation**

Parallel Region Example

```
#pragma omp parallel
{
    int tid = omp_get_thread_num();
    auto &mp = local_maps[tid];

    #pragma omp for schedule(static)
    for (int i = 0; i < n; ++i) {
        // Tokenization and counting
    }
}
```

---

5. Performance Considerations

- Using **local hash maps** reduces synchronization.
- `static` scheduling ensures balanced workload when lines are of similar size.

Reserving memory beforehand improves performance:

```
 mp.reserve(1 << 14);
freq.reserve(1 << 16);
```

-
- Merging maps is linear in total distinct words.

---

6. Expected Results

For a large text (e.g., Shakespeare plays, novels, logs), the parallel version should show:

- **Significant reduction in runtime**
- Correct total token count
- Correct top-N frequencies
- Higher scalability with more threads

---

7. Conclusion

This project has managed to show how OpenMP can be used effectively to parallelize a word-counting exercise. Implementation is scalable, does not create bottlenecks in synchronization and gets the same results.

It is a realistic demonstration of embarrassingly parallel workloads, which demonstrates how multi-core processors can radically enhance the performance of text analytics.