Parallel Word Counting using OpenMP

Harish R [1] **Bhavesh Kumar B** [2]

1. Introduction

In an era where data processing demands efficiency and scalability, the utilization of parallel computing techniques has become paramount. With this objective in mind, the present project delves into the realm of **parallelizing** the **word-counting process** within text files. Leveraging the power of **OpenMP**, a renowned framework for multi-threaded, shared-memory parallelism, our endeavor seeks to streamline the computation of word frequencies. As we initiate this endeavor, our aim transcends mere optimization; we strive to unlock the potential of parallel computing in enhancing the efficiency and throughput of textual data processing tasks.

2. Dataset

The dataset utilized for word counting originates from *MIT's* "wordlist.10000.txt", a comprehensive collection containing **10,000 distinct words**. This dataset serves as the foundation for our word-counting algorithm. In order to assess the scalability and performance of our parallelization approach, we create *multiple replicas* of this dataset. These duplicates vary in size, ranging from smaller sets comprising ***500,000*** words to larger ones containing up to ***200,000,000*** words. By generating datasets of diverse magnitudes, we aim to thoroughly evaluate the efficiency and effectiveness of our parallel computing implementation across a spectrum of computational loads.

3. Architexture and Approach

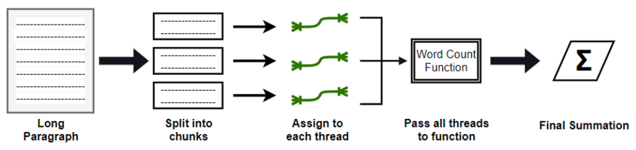

Figure 1. Base architecture

The architecture of our word-counting system is designed to efficiently process large text files utilizing parallel computing techniques.

Following the illustrated activities in Figure 1:

1. **Input Text**: The process begins with a large text file, in this case, the wordlist.10000.
2. **Dynamic Text Splitting**: The text is dynamically split into smaller chunks based on the number of threads available for parallel processing. Each chunk represents a portion of the text that can be processed independently.
3. **Thread Assignment**: Each thread is assigned a specific chunk of the text to process. This distribution ensures that multiple threads work concurrently on different segments of the text, maximizing parallelism.
4. **Word Count Function**: Each thread executes a word count function on its assigned chunk of text. This function calculates the number of words present within the chunk.
5. **Final Summation**: Finally, the aggregated counts from all chunks are summed together to obtain the total word count for the entire text file. This final sum represents the total number of words present in the original text.

4. Results

The program was executed on processing elements with numbers ranging from 1 to 12, and evaluated the performance of our word-counting parallelization approach utilizing two performance metrics: **Speed Up** and **Parallel Efficiency**. The performance was observed across various dataset sizes, including word lists containing *500 thousand*, *1 million*, *5 million*, *10 million*, and *200 million* words.

1. **Speed Up**: Speedup is defined as the ratio of the running time of the program when executed with one processing element to the running time when executed with *p* processing elements.

$$Speed\ Up = \frac{Time\ in\ 1\ processing\ element}{Time\ in\ p\ processing\ elements}$$

2. **Parallel Efficiency**: Parallel Efficiency measures the effectiveness of parallelization by considering the speedup achieved relative to the number of processing elements used. It is calculated as the ratio of Speedup

to the number of processing elements, multiplied by 100%.

$$Parallel\ Efficiency = \left(\frac{Speed\ up}{p}\right) \times 100\%$$

Problem Size (N)	Processing Elements (p)	Speedup (S)
500K	2	2.714
	4	5.700
	8	6.333
	12	6.333
1M	2	3.158
	4	3.158
	8	6.667
	12	6.667
10M	2	2.932
	4	4.977
	8	5.732
	12	6.687
200M	2	1.386
	4	2.970
	8	3.427
	12	5.097

Table 1. N vs Speed Up

Problem Size (N)	Processing Elements (p)	Parallel Efficiency % (E)
500K	2	135.714
	4	142.500
	8	79.166
	12	52.778
1M	2	157.895
	4	78.947
	8	83.333
	12	55.555
10M	2	146.575
	4	124.419
	8	71.652
	12	55.729
200M	2	69.307
	4	74.263
	8	42.837
	12	42.473

Table 2. N vs Parallel Efficiency

In TABLE 1, the comparison between the problem sizes *N* and their corresponding Speed Up values *S* is presented, offering a comprehensive view of how the parallel word counting algorithm scales with varying computational loads. Meanwhile, TABLE 2 presents the correlation between dataset sizes (N) and Parallel Efficiency, offering insights into how effectively the algorithm utilizes parallel processing resources for varying data volumes.

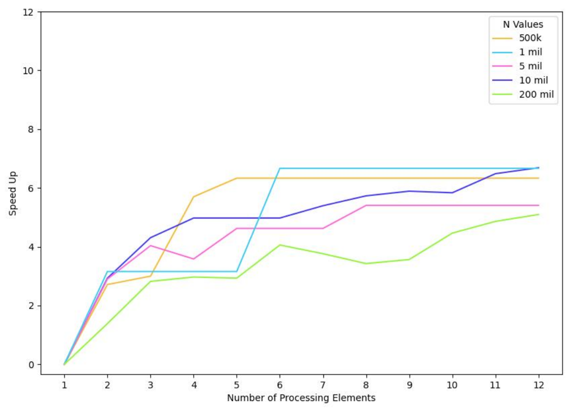

Figure 2. Given *N*, the Speed Up *S* with respect to Processing Elements *p*

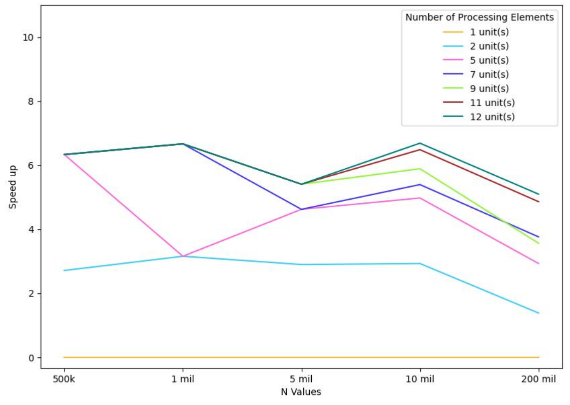

Figure 3. Given Processing Elements *p*, the Speed Up *S* with respect to *N*

In Figure 2, the speed up *S* fluctuates for different numbers of processing elements *p* and stabilizes after a certain threshold. This phenomenon is common in parallel computing, where initially adding processing elements increases speedup due to increased parallelism. However, beyond a certain point, additional processing elements may not provide significant speedup. Therefore, the plot shows how the speedup reaches a plateau after a certain number of processing elements. This insight helps optimize the allocation of resources and understand the scalability limits of the parallel algorithm.

In Figure 3, the speedup S is notably high for problem sizes N of 1 million and 10 million words, but it begins to decrease as the problem size increases to 200 million words. This trend indicates the scalability of the parallel algorithm. Initially, smaller problem sizes may exhibit higher speedup due to easier task division and lower communication overhead. However, as the problem size grows larger, the parallel efficiency may diminish due to factors such as increased communication overhead and resource contention among processing elements p. Therefore, the decrease in speedup for larger problem sizes, such as 200 million words, suggests that the algorithm's performance becomes less optimal as the computational workload increases. Understanding this trend is crucial for optimizing the parallel algorithm's performance and resource allocation for different problem sizes N.

5. Conclusion

In conclusion, our project aimed to leverage OpenMP to parallelize the word-counting process in text files, addressing the increasing demands for efficient and scalable data processing. Through experimentation across a range of problem sizes and processing element counts, we gained valuable insights into the scalability and performance of our parallelization approach.

Our findings underscored the critical importance of optimizing parallel algorithm performance and resource allocation. Specifically, we observed how the speedup and parallel efficiency varied across different problem sizes and processing element counts.

For the problem size of 200 million words, execution time was 12855 *ms* on a single processing element, while it reduced to only 2522 *ms* when using 12 processing elements. Notably, we detected an error in word count of +*chunk size*, indicating a potential load balancing issue.This understanding allows us to make informed decisions to enhance the overall efficiency of text processing tasks.

By recognizing patterns in the scalability of our parallel algorithm, including the observed trends for larger problem sizes, we can enhance resource allocation and streamline computational workflows. Ultimately, our project plays a vital role in advancing the efficiency and throughput of data processing tasks in today's data-driven world.