OPTIMIZING K-MER COUNTING IN LARGE SCALE GENOMIC DATASETS USING MAPREDUCE

JACOB IOFFE AND PARWAZ GILL

CS 5266

Introduction:

Genomic analysis has become an essential tool for understanding the structure, function, and evolution of DNA sequences. As the field of computational genomics continues to develop, efficient techniques for analyzing large volumes of sequencing data are crucial. One such technique is k-mer counting, which provides valuable insights into the frequency of specific DNA sequences. This paper will provide an overview of k-mers, FASTA files, the importance of k-mer counting in computational genomics, and its real-world applications. Furthermore, we will discuss our project, which employs the MapReduce framework to parallelize the k-mer counting process from DNA sequencing data stored in the Maizie Zhou Lab.

A k-mer is a continuous substring of length k extracted from a DNA sequence. By counting the occurrences of k-mers, researchers can identify unique sequences within a genome and distinguish between closely related species or strains. K-mer counting is essential in computational genomics, as it allows for efficient sequence comparison, genome assembly, and functional annotation. The K-mers we are working with consist of the human chromosome complex and are stored in FASTA files. These files contain millions of short DNA sequences generated by sequencing machines, providing the raw data necessary for k-mer analysis.

K-mer counting has a wide range of real-world applications, from understanding the genetic basis of diseases to the development of new drugs and therapies. By efficiently analyzing DNA sequences, researchers can identify potential target genes, detect variations between populations, and better understand evolutionary relationships.

In this paper, we present our project that uses the MapReduce framework to parallelize the k-mer counting process and explore the enhancement of the speed and efficiency of genomic analysis. By comparing two different implementations of k-mer counting, we aim to optimize the process and provide a scalable solution for processing vast amounts of DNA sequencing data. Through our experiments, we hope to contribute to the growing field of computational genomics and provide valuable insights for researchers working with large-scale DNA sequence data.

Methodology:

The Data

The data is a DNA sequence of the human chromosome complex, and the primary goal of our project is to extract k-mers from these sequences to identify unique sequences within a genome and distinguish between closely related species or strains. Specifically, we are working with the In our study, we utilize the dataset NA24385 illumina hg19 chromosome, which comprises Illumina sequencing data from the human sample NA24385, a sample sourced from the 1000 Genomes Project. The 1000 Genomes Project is an international initiative aimed at cataloging human genetic variation by sequencing the genomes of over 2,500 individuals. The sequencing data in this dataset has been aligned to the hg19 reference genome assembly, which serves as a standardized coordinate system for annotating and analyzing genomic data. Furthermore, our dataset for the scope of the project specifically focuses on chromosome 1 of the human genome, which is the largest human chromosome, encompassing approximately 8% of the entire genome. This dataset provides an ideal resource for our project, as it offers high-quality sequencing data for testing and optimizing our k-mer counting methodology.

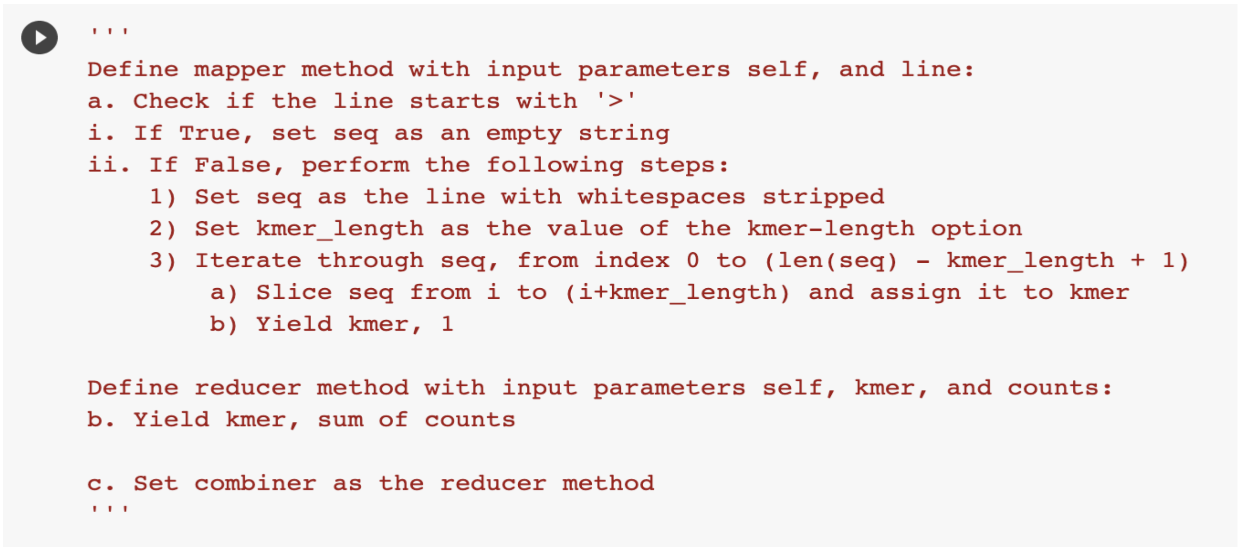
While the entire chromosome is roughly 200GB, in order to test the MapReduce in our learner lab environments, we aimed to explore chromosome 1 and cut the data to 1 GB for testing and time.

Design of the solutions

In this study, we aimed to compare the efficiency and performance of two distinct MRJob implementations, KmerCount (a) and KmerCount (b), designed for counting the number of k-mers in a given FASTA file. The primary distinction between the two implementations lies in their mapper functions. While KmerCount (a) employs a straightforward loop to iterate through the sequence and yield k-mers for each position without any optimizations, KmerCount (b) incorporates an optimized string slicing technique in its mapper function.

For KmerCount (a), the mapper function adopts a simplistic approach by iterating through the DNA sequence and yielding k-mers for each position. This implementation does not incorporate any optimization strategies, which may impact the overall performance of the k-mer counting process.

In contrast, KmerCount (b) implements an optimized string slicing technique within its mapper function. During the iteration process, the function checks whether the previous character in the DNA sequence is equal to the last character of the current k-mer. If the characters are not equal, the k-mer is sliced and yielded. However, if the characters are equal, the same k-mer is yielded without the need for slicing. Both function pseudocodes are seen in Figures 1 and 2.

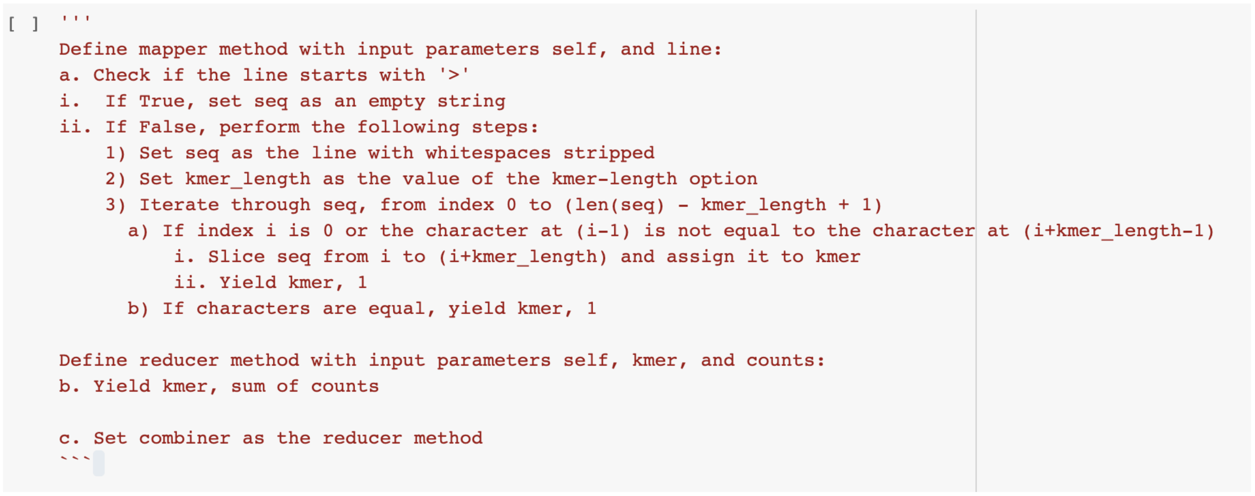


```
'''
Define mapper method with input parameters self, and line:
a. Check if the line starts with '>'
i. If True, set seq as an empty string
ii. If False, perform the following steps:
    1) Set seq as the line with whitespaces stripped
    2) Set kmer_length as the value of the kmer-length option
    3) Iterate through seq, from index 0 to (len(seq) - kmer_length + 1)
        a) Slice seq from i to (i+kmer_length) and assign it to kmer
        b) Yield kmer, 1

Define reducer method with input parameters self, kmer, and counts:
b. Yield kmer, sum of counts

c. Set combiner as the reducer method
'''
```

Figure 1: Kmercount (a) Pseudocode



```
[ ] '''
    Define mapper method with input parameters self, and line:
    a. Check if the line starts with '>'
    i.  If True, set seq as an empty string
    ii. If False, perform the following steps:
        1) Set seq as the line with whitespaces stripped
        2) Set kmer_length as the value of the kmer-length option
        3) Iterate through seq, from index 0 to (len(seq) - kmer_length + 1)
          a) If index i is 0 or the character at (i-1) is not equal to the character at (i+kmer_length-1)
              i. Slice seq from i to (i+kmer_length) and assign it to kmer
              ii. Yield kmer, 1
          b) If characters are equal, yield kmer, 1

    Define reducer method with input parameters self, kmer, and counts:
    b. Yield kmer, sum of counts

    c. Set combiner as the reducer method
    ```
```

Figure 2: Kmercount (b) Pseudocode

**Technologies:**

In our study, we utilized various technologies to efficiently manage and process large-scale DNA sequencing data. Amazon S3 (Simple Storage Service) is a cloud-based object storage service that enables storing and retrieving data from anywhere on the web. We leveraged S3 buckets to store and access the DNA sequencing data for our project.

Furthermore, we employed Google Colab, a cloud-based platform for executing Python code, which offers a free, GPU-accelerated environment suitable for data analysis, machine learning, and other computational tasks. By utilizing Google Colab, we were able to write MR jobs and effectively use the MapReduce framework to parallelize the k-mer counting process.

Lastly, we executed the MR jobs on an Amazon EMR (Elastic MapReduce) cluster. Amazon EMR is a web service that facilitates running big data frameworks, such as Apache Hadoop and Apache Spark, on AWS (Amazon Web Services). Running our MR jobs on an EMR cluster (Hadoop) provided us with a scalable, managed infrastructure that could handle the
```

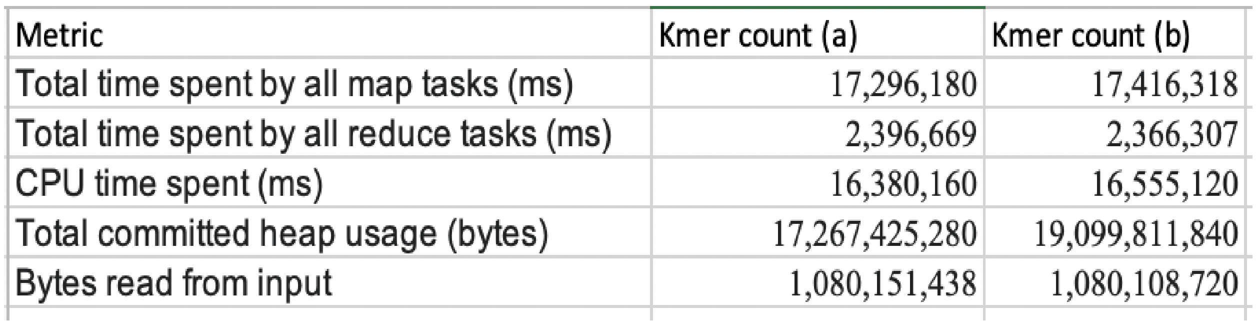
processing of vast amounts of DNA sequencing data, which was crucial to the success of our project.

Results:

KmerCount (b), which uses the optimized string slicing technique, shows slightly higher total time spent by all map tasks (17,416,318 ms) compared to KmerCount (a) (17,296,180 ms).However, this difference in performance may be attributed to various factors such as system load, data distribution, or even the order of data processing.

The reduce function implementations for both KmerCount (a) and KmerCount (b) are identical. KmerCount (b) has a slightly lower total time spent by all reduce tasks (2,366,307 ms) compared to KmerCount (a) (2,396,669 ms). This difference in performance can also be attributed to similar factors as mentioned in the mapper function performance. KmerCount (b) utilizes more heap memory (19,099,811,840 bytes) compared to KmerCount (a) (17,267,425,280 bytes). The difference in heap memory usage could be due to variations in data distribution or the internal workings of the memory management system.

KmerCount (a) reads slightly more bytes from the input (1,080,151,438 bytes) than KmerCount (b) (1,080,108,720 bytes). This difference in input data size may influence the performance metrics observed, although it's a relatively small difference.



Metric	Kmer count (a)	Kmer count (b)
Total time spent by all map tasks (ms)	17,296,180	17,416,318
Total time spent by all reduce tasks (ms)	2,396,669	2,366,307
CPU time spent (ms)	16,380,160	16,555,120
Total committed heap usage (bytes)	17,267,425,280	19,099,811,840
Bytes read from input	1,080,151,438	1,080,108,720

Table 1: Results from MR Job

Conclusion:

KmerCount (b) uses an optimized string slicing technique that checks for equality between adjacent characters and avoids unnecessary slicing operations. The performance metrics show only slight differences between the two implementations, which may be influenced by various factors such as system load, data distribution, or the order of data processing. While KmerCount (b) demonstrates a slightly higher total time spent by all map tasks, it also has a slightly lower total time spent by all reduce tasks, which suggests that the optimized string slicing technique may have a minimal impact on overall performance. Further experimentation and performance analysis can be conducted with different datasets and varying kmer lengths to determine the benefits of the optimized string slicing technique in KmerCount (b) compared to the non-optimized implementation in KmerCount (a).

Further Analysis:

To get a better understanding of the impact of the optimized string slicing technique, we could consider running the jobs with varying kmer lengths and different dataset sizes. This will help in identifying any trends or patterns in the performance differences.

We could investigate the impact of different Hadoop configurations, such as the number of reducers, mappers, and the amount of memory allocated to the tasks, to understand how the optimized string slicing technique performs under different conditions.

We could also analyze the scalability of both implementations by running them on larger and more complex datasets to evaluate if the optimized string slicing technique provides significant benefits for larger-scale problems.

Further, we could consider other optimization techniques, such as using more efficient data structures or parallelization strategies, to improve the overall performance of the KmerCount job. At last, we could explore alternative big data processing frameworks, such as Apache Spark or Flink, to compare their performance against Hadoop MapReduce and assess if they offer any advantages for this specific problem.