Applying Concurrency Technique using Golang in Multiple Approximate Pattern Matching Problem with Burrows-Wheeler Transform

Ta Van Nhan
Faculty of Mathematics, Mechanics, Informatics,
VNU University of Science,
Hanoi, Vietnam.
tavannhan@gmail.com

Nguyen Thi Hong Minh
Faculty of Mathematics, Mechanics, Informatics,
VNU University of Science,
Hanoi, Vietnam.
minhnth@gmail.com

***Abstract*—Sequence alignment is a process that look for matches, mismatches and gaps between DNA sequence-reads and the reference genome, which requires a lot of time and memory. For large, complex, and diverse data like the human genome, there are many methods to solve this problem, among which the group of method based on the Burrows-Wheeler Transform (BWT) have shown many advantages.**

In this paper, we applied the concurrency technique within the alignment algorithm with Burrows-Wheeler Transform and implemented in Golang. Execution of the program shows time efficiency in working simultaneously. Moreover, we also offered a balance between running time and working space in order to ensure the goal and to consistent with the capacity of computing system.

***Index Terms*—Multiple Approximate Pattern Matching, SNPs, Burrows-Wheeler Transform, Sequence Alignment, Concurrency Technique, Golang.**

I. Introduction

DNA sequencing is a process to build a sequence of nucleotides (A (Adenine), T (Thymine), G (Guanine), C (Cytosine)) of a DNA molecule. The sequence determined in special segments (called genes) is the basis to help us understand the biological mechanism of living organisms. Depending on different purposes, a whole genome, or a whole exome, or a target is selected for sequencing. Sequencing methods has become popular and easy to implement with the development of next-generation sequencing technologies and the support of computational techniques. During the sequencing process, reads may not exactly match to the reference genome (called approximate patterns or inexact patterns) if there is a mutation or an error of the sequencing machine [1]. Small genetic alterations such as single nucleotide polymorphisms (SNPs), insertions/deletions (indels) occur frequently in the DNA. These can be identified by detecting mismatches or gaps after the sequence alignment. Heng Li and Richard Durbin (2009) showed how to effectively search mismatches and gaps, by using the alignment algorithm with the Burrows-Wheeler Transform [2]. Instead of the original sequence, the algorithm works on the string created from the last column of the matrix made up of *Suffix Arrays*. The string conversion method in terms of block sorting for the lossless data compression algorithm was proposed by Michael Burrows and David Wheeler (1994) [3]. Here, *Suffix Trees* is replaced by *Suffix Arrays* to significantly improve memory efficiency [4]. Furthermore, it consumes less memory by storing only *Partial Suffix Arrays*. This is one of two components to balance between the runtime and memory usage. The other element lies in the process of finding approximate patterns on *Checkpoint Arrays* [5]. The sequence alignment algorithm based on the Burrows-Wheeler Transform could be further accelerated when executed by a concurrent programming language.

Recently, Golang is emerging as a powerful programming language for processing and computing big data. Its most powerful feature is its concurrency function, which automatically exploits the operation of the computer's core without depending on the management and allocation of the operating system. In Golang, the threads on the cores are implemented by goroutines with many advantages. Each goroutine only uses very little memory from the heap (only about 2kB[1]), which makes it possible to generate lots of goroutines at the same time. Goroutines communicate with each other quite simply by transmitting any type of data and they can work concurrently on logical processors at exactly the same time. In addition, they can wait for each other in a queue. Since goroutines operate on the principle of none-sharing variables, one needs to synchronize the communication of goroutines by using buffered channels or unbuffered channel [6]. Moreover, the execution efficiency also depends on how the algorithm is designed with CPU-bound or IO-bound workloads [7]. When applying Golang to the alignment algorithm, we can use both designs when the goroutines work in parallel, or when they have to move in and out of a queue.

In this paper, we focused on designing algorithms on the

This research was partially sponsored by VinIF in the scope of the cooperation trainning program of Master in Data Science at the Faculty of Mathematics, Mechanics and Informatics, VNU University of Natural Science.

[1]https://golang.org/doc/go1.4

basis of the Burrows-Wheeler Transform algorithm improvement to achieve the highest efficiency when deploying and taking advantage of Golang's strengths. We hope that this is an approach for testing the new programming language that has advantages in speed, popularity, and ease of installation on universal computing systems for the bioinformatics problem in general, and the problem of sequence alignment in particular. The salient feature of the sequencing-related algorithms is that the results of a step usually depend on the results in the previous step, so in addition to the parts that can be parallelized, there would be sequential forced parts. Our innovation lies in maximizing the possible parts and optimizing the time of context switching when goroutines have to wait for each other. The algorithm test results will be compared with those of BWA-MEM tool, the tool being developed in C language with high accuracy and very fast speed [8]. However, a challenge to this is that the algorithm used in BWA-MEM has not been published yet, so we could only compare the sequence matching results on the same dataset. Besides, based on the characteristics of the algorithm, we also proposed a method of setting the parameters to adjust the increase or decrease between the computation time and the memory required.

II. Preliminary

A. Suffix Arrays

Given that the string T includes only letters and ends with \$, which is the character preceding the letters in the charset.

- *Suffix String* (SS) is the substring starting from somewhere in the string T and ends at \$.
- *Suffix Matrix* (SM) is a matrix where the rows are lexicographically ordered suffix strings.
- *Suffix Arrays* (SA) is an array storing the beginning positions of each *Suffix Strings* in the *Suffix Matrix*.

Hon et al. (2007) published an efficient algorithm to build *Suffix Arrays* of length $|T|$ with $\mathcal{O}(|T|.log(|T|))$ runtime using $\mathcal{O}(|T|.log(|\Sigma|))$ bits of memory usage, where $|\Sigma|$ is the unique characters in the string T, excluding the charater \$ [9]. To reduce memory requirements, we could simply store *Partial Suffix Arrays* [5].

B. Burrows-Wheeler Transform

Having obtained *Suffix Arrays* SA of the string T, one could construct the Burrows-Wheeler Transform BWT of the string T with linear time [9]. Denote BWT_i, T_i by the i^{th} symbols of BWT and T respectively, SA_i is the value at i^{th} position of *Suffix Arrays*, one have:

$$BWT_i = \begin{cases} T_{SA_i-1} & \text{if } SA_i > 0 \\ T_{|T|-1} & \text{if } SA_i = 0 \end{cases}$$

for i = 0, 1, ... , $|T| - 1$.

The Burrows-Wheeler Transform Matrix ($BWTM$) could be obtained from the matrix SM by adding each prefix string to the end of each *SS* correspondingly. Considering the example in figure 1, the matrix $BWTM$ and SA are generated from the string ATCATGATC\$. It can be easily seen that the string BWT is the last column of the matrix $BWTM$. Furthermore, sorting BWT column by alphabetical order would yield the first column of the matrix $BWTM$. In the following operations of the paper, we would consider these two columns of the matrix $BWTM$. The first column is denoted by FC and the last column is BWT.

BWTM	Index	SA
\$ATCATGAT**C**	0	9
ATC\$ATCAT**G**	1	6
ATCATGATC**\$**	2	0
ATGATC\$AT**C**	3	3
C\$ATCATGA**T**	4	8
CATGATC\$A**T**	5	2
GATC\$ATCA**T**	6	5
TC\$ATCATG**A**	7	7
TCATGATC\$**A**	8	1
TGATC\$ATC**A**	9	4

Fig. 1. The matrix $BWTM$ and array SA of the string ATCATGATC\$.

C. Checkpoint Arrays

After construction the string BWT of the sequence T, the location of a symbol of the string BWT could be found on sequence T. To do this, one needs to count the number of occurrences of a character in the string BWT from location 0 to any location within the length of BWT. To save time, these count values are stored in *Checkpoint Arrays*. To save memory, may simply store a subset of the arrays [5].

The example in figure 2 is *Checkpoint Arrays* of the Burrows-Wheeler Transform CG\$CTTTAAA obtained from the string ATCATGATC\$.

Index	BWT	\$	A	T	C	G
0	C	0	0	0	1	0
1	G	0	0	0	1	1
2	\$	1	0	0	1	1
3	C	1	0	0	2	1
4	T	1	0	1	2	1
5	T	1	0	2	2	1
6	T	1	0	3	2	1
7	A	1	1	3	2	1
8	A	1	2	3	2	1
9	A	1	3	3	2	1

Fig. 2. *Checkpoint Arrays* store the occurrences of symbols belonging to the string BWT from position 0 to any position less than or equal to the length of string BWT.

III. Methods

A. Exact Pattern Matching

P. Ferragina and G. Manzini (2005) introduced a backward search algorithm that counts the occurrences of a pattern P on

the string T with $\mathcal{O}(|P|+|T|)$ runtime [11]. In this algorithm, each symbol from the end to the beginning of the pattern is compared to each symbol in an interval of the column BWT that corresponds to a previous interval of the column FC. One denote the first position of $symbol$ appearing in FC by $FO(symbol)$, and $C(symbol, i)$ is the number of times that $symbol$ has appeared in BWT from position 0 to position i^{th}. From an interval in the column BWT, one could find a new interval [top, $bottom$] in the column FC, the intervals are updated as follows:

$$top \leftarrow FO(symbol) + C(symbol, top - 1)$$
$$bottom \leftarrow FO(symbol) + C(symbol, bottom) - 1$$

Matching positions of the pattern P on the string T are equal to the value of *Suffix Arrays* SA in the order corresponding to the positions on FC in the last step.

Here, the function $C(symbol, i)$ can be replaced by calling the character count value of $symbol$ stored in the *Checkpoint Arrays* to speed up the algorithm. An example of backward searching for the pattern ATC in the string ATCATGATC (Figure 3, (I)) shows that the pattern matches the given string at positions 0 and 6.

In the case of using *Partial Suffix Arrays*, one continues to backwardly search for positions in the interval [top, $bottom$] until these positions are present in the arrays. Then, values of matching positions are equal to corresponding values present in *Partial Suffix Arrays* plus a number of backward steps. As with the above example, suppose one only has a *Partial Suffix Arrays* with a space of 4 units (Figure 4). The position 2 corresponding to the value 0 exists in *Partial Suffix Arrays*, the position 1 has no corresponding value in the array, so one need to continue the backward search. This process takes 2 more steps before finding the value 4 in the arrays, the corresponding match position of the pattern on the given string is: $4+2=6$.

B. Approximate Pattern Matching

For exact pattern matching, symbols of a pattern should all match symbols of the sequence. However, symbols of an approximate match pattern could differ from symbols in the sequence, provided number of differences is less than an allowable difference threshold. In addition, the exact pattern match algorithm is still applied for finding an approximate match pattern. For example, the pattern ATC approximately matches to the sequence ATCATGATC at the position 3 with the difference threshold 1 (Figure 3, (II)). On the other hand, there are intervals [top, $bottom$] in FC where one can not find an approximate match pattern with an allowable threshold (Figure 3, (III)).

In order not to spend time searching for a case that will not satisfy, one can place a lower bound for differences in *Difference Arrays* that can be identified through the process of comparing substrings of the pattern P with the reverse string of T (Algoritm 1). When top is greater than bottom, there is a difference between the two strings, thus the difference could be a mismatch or a gap. The reason for using the inverse string of T is that one uses backward search. *Difference Arrays* is used primarily for the sequential algorithm (Algoritm 2).

Algorithm 1 Difference Array Calculating

Input: BWT': Burrows - Wheeler Transform of Reverse String, PAT: Pattern, FO: First Occurrence, C': Checkpoint Arrays of BWT'.
Output: D.
Initialisation: $N, t, bt \leftarrow |PA|, 1, |BWT'|$; $z \leftarrow 0$

DAC (BWT', PAT, FO, C')
1: **for** $i \leftarrow 1$ to N **do**
2: $symbol \leftarrow PAT[i]$
3: $nT \leftarrow FO(symbol) + C(symbol, t-1)$
4: $nB \leftarrow FO(symbol) + C(symbol, bt) - 1$
5: **if** $t > bt$ **then**
6: $t, bt \leftarrow 1, |BWT^{'}|$
7: $z \leftarrow z + 1$
8: **end if**
9: $D(i) \leftarrow z$
10: **end for**
11: **return** D

Algorithm 2 Sequential Approximate Pattern Matching

Input: BWT: Burrows-Wheeler Transform, D: Difference Arrays, PSA: Partial Suffix Arrays, W: Difference Threshold; PAT, FO, C.
Output: ML: Match Location, NM: Number of Mismatch.
Initialisation : $tbtUpdate \leftarrow (1, |BWT|, W, |PAT|)$.
1: **while** $|tbtUpdate| \neq 0$ **do**
2: $tbt, t, bt, d, id \leftarrow tbtUpdate[1], tbt[1:4]$
3: Remove tbt from $tbtUpdate$
4: $apr \leftarrow (A, C, G, T)$
5: **if** $id \neq 1$ **then**
6: $PAT, symbol \leftarrow PAT[1:id], PAT[id]$
7: **for** $i \leftarrow 1$ to $|apr|$ **do**
8: $aprS, nD, nT, nB \leftarrow apr[i], d, t, bt$
9: **if** $nD \geq D[id]$ **then**
10: **if** $aprS \neq symbol$ **then**
11: $nD \leftarrow nD - 1$
12: **end if**
13: $nT \leftarrow FO(aprS) + C(aprS, t-1)$
14: $nB \leftarrow FO(aprS) + C(aprS, bt) - 1$
15: **if** $nT \leq nB \cap nD \geq 0$ **then**
16: Append $(nT, nB, nD, id-1)$ to $tbtUpdate$
17: **end if**
18: **end if**
19: **end for**
20: **else**
21: Get all nT, nB, and nD from $tbtUpdate$
22: $ML, NM \leftarrow$ values of $PSA[nT:nB]$, $W - nD$
23: **return** ML, NM
24: **end if**
25: **end while**
26: **return** Pattern doesn't match to sequence.

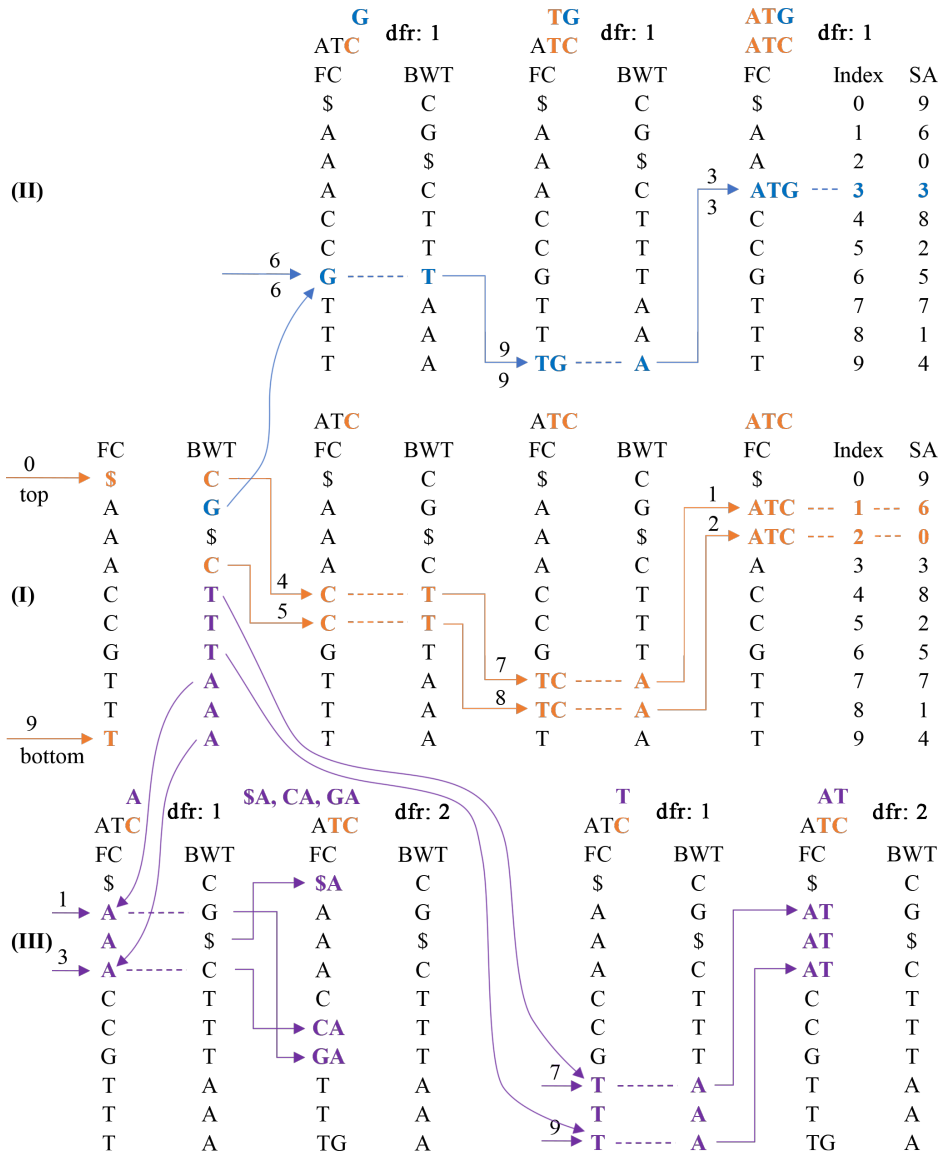

Fig. 3. The backward search for the pattern ATC in the string ATCATGATC. (I) The pattern exactly matches the given string at positions 0 and 6. (II) The pattern approximately matches the given string at the position 3 with the difference threshold 1. (III) The pattern does not match to the given string because number of differences is over the difference threshold 1.

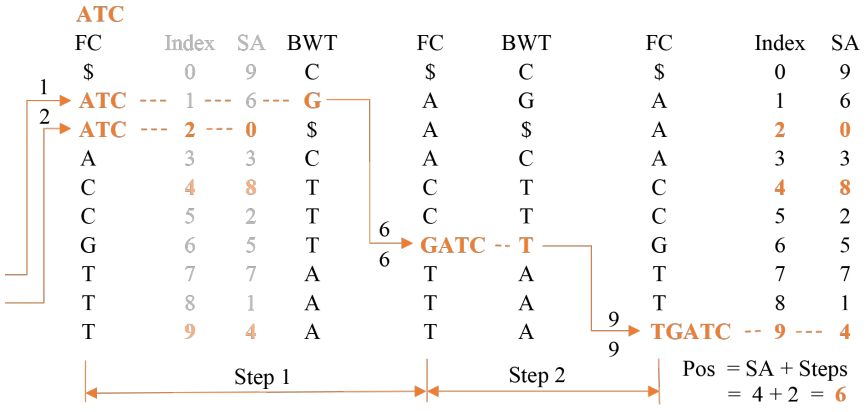

Fig. 4. Continue backward search for the pattern ATC if there is only *Partial Suffix Arrays* that its values separated by 4 units. Finally, one find the match position that is 6.

C. Applying Golang to Approximate Pattern Matching

In Golang, it is possible to divide a number of goroutines equal to or greater than a number of logical processors of a computer. When the goroutine number equals the logical processor number, the execution time is equal to the running time of the slowest goroutine. On the other hand, when it is larger, goroutines would be synchronized to optimize time and to ensure that no core is idle.

If an array is used to store updated intervals [top, $bottom$] after each while loop in the sequential algorithm (Algorithm 2), then an adequate amount of goroutines are used to perform the update in the concurrency algorithm (Algorithm 3). Concurrently, buffered channels are also initialized and they act as an intermediary to receive and send data between goroutines. Note that buffered channels could send data to different goroutines at the same time, but they do not store the sent data. Therefore, one does not need to delete old intervals [top, $bottom$] after calculation. These are the basis for synchronizing the working process of goroutines through buffered channels.

- Line 1: the for loop means simultaneously setting goroutines with a given quantity.
- Line 3: goroutines receive data of intervals [top, $bottom$] at the same time from buffered channel, the data is lost immediately on the channel.
- Line 23: the data of a new interval [top, $bottom$] is sent back to the previous buffered channel.
- Line 30: once a read has been browsed, the data about the last interval [top, $bottom$] is sent into a new buffered channel, where one could find the match positions of the pattern.
- Line 33: if the buffered channel is empty, one exits the loop to avoid "deadlock" when the data is invoked by a goroutine from an empty channel, which also means that no approximate reads could be found.

Here, goroutines are working on the same task. One could adapt the design so that they can take on different tasks. Lines 4 through 7 are the process of assigning values to variables, it does not take a long time. In line 8, new goroutines take tasks to update new intervals [top, $bottom$] corresponding to the symbols of the set $\{A, C, G, T\}$ derived from an interval [top, $bottom$]. Notice that there are 4 elements of the set $\{A, C, G, T\}$, so there are not too many goroutines required resulting in the fact that they have to wait for each other, the goroutine number is chosen by 4. After these goroutines have completed their duties, the original goroutine resumes the rest. In this algorithm, since the child goroutines are created in one parent goroutine, even though they do different tasks, the parent goroutine still has to wait for its offspring to finish. If one compares these offspring with other parent goroutines that do not contain it and concurrently work with it, it is clear that they do different tasks at the same time.

Algorithm 3 Concurrency Approximate Pattern Matching

Input: BWT, PAT, PSA, FO, C, D, W,
GS: Number of Goroutines.
Output: ML, NM.
Initialisation:
$posChan, tbtUpchan \leftarrow$ buffered channel,
$tbtUpChan, gs \leftarrow (1, |BWT|, W, |PAT|), 4$

CAPM (BWT, PAT, PSA, FO, C, D, W, GS)

```
1:  for g = 1 to GS do
2:    go func()
3:    for Receive tbt from range of tbtUpChan do
4:      t, bt, d, id ← tbt[1], tbt[2], tbt[3], tbt[4]
5:      apr ← (A, C, G, T)
6:      if id ≠ 1 then
7:        PAT, symbol ← PAT[1 : id], PAT[id]
8:        for g ← 1 to gs do
9:          if tbt = (0, 0, 0, 0) then
10:           Send (0, 0, 0, 0) to tbtUpChan
11:           break
12:         end if
13:         go func(g)
14:         aprS, nD ← apr[g], d
15:         nT, nB ← t, bt
16:         if nD ≥ D[id] then
17:           if aprS ≠ symbol then
18:             nD ← nD − 1
19:           end if
20:           nT ← FO(aprS) + C(aprS, t − 1)
21:           nB ← FO(aprS) + C(aprS, bt) − 1
22:           if nT ≤ nB ∩ nD ≥ 0 then
23:             Send (nT, nB, nD, id − 1) to tbtUpChan
24:           end if
25:         end if
26:         end go func(g)
27:       end for
28:     else
29:       if tbt ≠ (0, 0, 0, 0) then
30:         Send tbt to posChan
31:       end if
32:     end if
33:     if |tbtUpChan| = 0 then
34:       for i = 1 to GS do
35:         Send (0, 0, 0, 0) to tbtUpChan
36:       end for
37:       break
38:     end if
39:   end for
40:   end go func()
41: end for
42: if |posChan| ≠ 0 then
43:   close(posChan)
44:   Receive nT, nB, and nD from range of posChan
45:   ML, NM ← values of PSA[nT: nB], W − nD
46: end if
47: return ML, NM
```

The working process of goroutines is quite complicated, so it is very easy to raise ”all goroutines are asleep - deadlock!” if a goroutine receives data from an empty channel. To avoid this, one need to send additional elements to the buffered channel. A number of elements (0, 0, 0, 0) equal to the number of parent goroutines can be added to the channel at line 35 (Algorithm 3), which does not participate in the update process of intervals [top, $bottom$] and is removed from the result if they appear, but one still need to maintain their original count in the channel.

Returning to the example about approximate matching of the pattern ATC to the string ATCATGATC, one would observe the process of updating the intervals [top, $bottom$] with the sequential algorithm and the concurrency algorithm (Figure 5). With sequential algorithm, we start searching from the interval [0, 9]. Since the search is based on comparing symbols of the pattern with symbols of the string BWT, new intervals are found one after another in a constant order as follows [1, 3], [4, 5], [6, 6], and [7, 9]. For the concurrency algorithm, we cannot know the order in advance, when a certain parent goroutine takes the task of calculating with the interval [0, 9], its child goroutines in turn search for the corresponding new chunk of symbols in BWT. Note that this concurrency prevents us from knowing which symbols would be pre-selected for backward search, which means that we do not know which new intervals would be found first. Each execution of the algorithm will produce a different order, in this example the order of new intervals are [6, 6], [7, 9], [4, 5], and [1, 3].

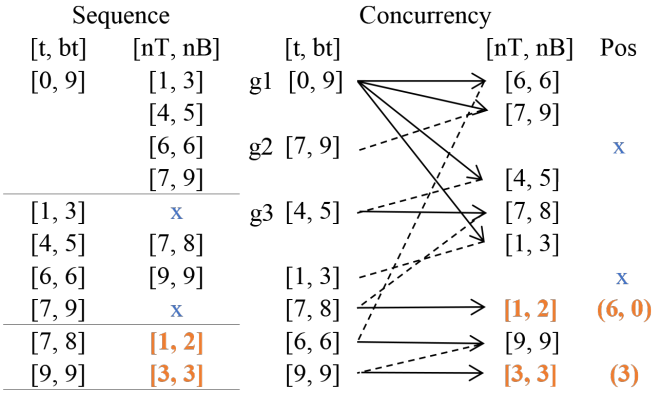

Fig. 5. Considering the process of approximate matching of the pattern ATC to the string ATCATGATC with the sequential algorithm and the concurrency algorithm.

Let us now consider only the work of parent goroutines. When goroutine 1 finds a new interval [7, 9], it is also time for goroutine 2 to begin further calculations with this chunk just like the task of goroutine 1. While goroutine 1 continues to search, goroutine 2 ends the task because the number of mismatches has exceeded 1. Shortly thereafter, goroutine 1 finds another new interval [4, 5], then goroutine 3 begins to accept its task with the interval. And while goroutine 1 was still working, goroutine 3 finds a new interval [7, 8]. The session of goroutine 1 ends when it is able to update data of interval [1, 3], now it could work with another interval or wait for its turn. As such, these parent goroutines work simultaneously, but do not start and end at the same time. The goroutines are synchronized to ensure the fastest possible tasks. This also avoids the idle state of any logical processor during system execution.

In practice, the number of reads generated by a sequencing platform is usually large, but the number remains unchanged during the operation of the algorithm. The workload may vary from read to read but it remains similar if one views on equally divided parts of all reads. This is the CPU-bound workload, if each goroutine computes a part that is evenly divided, they can work in parallel, independently at the same time, and no waiting states are generated during the work. For this type of workload, the number of goroutines is set to the correct number of logical processors (usually 2 times the number of cores) of the computer to make the most of the system's computation time (Algorithm 4).

IV. Implementation

A. Data

The COVID-19 pandemic which started in early 2020 has become a global major public health crisis of the current year. This is caused by an outbreak of a virus belonging to a family of viruses called coronavirus, namely SARS-CoV-2, which causes the ”Severe Acute Respiratory Syndrome” (SARS) [5]. SARS-CoV-2 penetrates human cells through Angiotensin converting enzyme 2 (ACE-2) attached to the cellular membranes of cells in the lungs, arteries, heart, kidneys and intestines. It then attacks the Antigen-presenting cells (APC) of the immune system, while reducing T cells (a type of Lymphocyte - a subclass of Leukocytes) [12]. The main reason for the fast mutational rate of a virus is that its genetic material is RNA instead of DNA, whose replication process is more prone to errors compared to DNA replication. Therefore SARS-CoV-2 in particular and coronaviruses in general change quickly and unpredictably.

We retrieved the raw sequences of SARS-CoV-2 published on July 28, 2020 by KwaZulu-Natal Research Innovation and Sequencing Platform from the Sequence Read Archive (SRA)[2]. The FASTQ file includes 436,610 paired-end reads [13]. The FASTQ file was converted to the fasta file (named ”Sra_SARs_CoV_2.fasta”) by the tool FASTQ to FASTA converter on Galaxy Version 1.1.5 [14]. The genome assembly of SARS-CoV-2 published by Fan Wu et al. (2020)[3] [15], which is 24748 bp long was used as the reference genome for alignment. The reference genome file was renamed to ”Ref_SARs_CoV_2.fa”.

[2]https://sra-pub-sars-cov2.s3.amazonaws.com/sra-src/SRR12338312/KPCOVID-345_S81_L001_R1_001.fastq.gz.1

[3]https://www.ncbi.nlm.nih.gov/nuccore/NC_045512.2

Algorithm 4 Concurrency Multiple Approximate Pattern Matching

Input: BWT, $PATS$, PSA, FO, C, D, W, GS.
Output: ML, NM.
Initialisation : $GS \leftarrow$ number of logical processors

CMAPM $(BWT, PATS, PSA, FO, C, D, W, GS)$
1: $st \leftarrow |PATS|/GS$
2: **for** $g \leftarrow 1$ to GS **do**
3: **go func(g)**
4: $start \leftarrow g * st$
5: $end \leftarrow start + st$
6: **if** $g = GS$ **then**
7: $end \leftarrow |PATS|$
8: **end if**
9: **for** $i \leftarrow start$ to end **do**
10: $PAT \leftarrow PATS[i]$
11: $ML, NM \leftarrow$ MP($BWT, PAT, PSA, FO, C, D, W, GS$)
12: Send ML, NM to ch
13: **end for**
14: **end go func()**
15: **end for**
16: **close** ch
17: Receive ML, NM from ch
18: **return** ML, NM

B. Parameters and Inputs

The Burrows-Wheeler Transform, Suffix Arrays, and *Checkpoint Arrays* were first pre-computed prior executing the algorithm. As described in the section II , we change the distance c between values in *Suffix Arrays* and the distance k between the order in *Checkpoint Arrays* to evaluate the variation level between runtime and memory used. Next, we executed the algorithm with values of difference and compare the results with one output from BWA-MEM tool. These results are identical when parameters of BWA-MEM tool is set as follows:

- T (do not export alignments with scores lower than an integer) = 0.
- k (matches shorter than an integer are removed) = 0.

TABLE I
PARTIAL SUFFIX ARRAYS AND CHECKPOINT ARRAYS

	Partial Suffix Arrays		Checkpoint Arrays	
c, k	***Time (ms)***	***Memory (MiB)***	***Time (ms)***	***Memory (MiB)***
1	18.64	2	302.09	13
30	15.26	2	10.59	4
60	15.18	2	4.79	3
100	14.00	2	2.89	2

TABLE II
TOTAL OF MATCHES WITH DIFFERENCE THRESHOLDS

D	Time (s)	Memory (MiB)	Total
0	178.56	96	242 943
1	234.31	270	354 930
2	436.03	268	365 724
3	1229.37	357	367 946

C. Results

We use a virtual machine with 8 vCPUs and 52 GB of memory on Google Cloud platform to implement the algorithm. For *Partial Suffix Arrays*, there is no significant difference between runtime and working space when the values c and k are different. For *Checkpoint Arrays*, the measurement runtime is 104.5 times slower and the memory requirement is 6.5 times greater when c and k are 1 compared when c and k are 100 (see Table I). However, it is only necessary to run the algorithm to find *Checkpoint Arrays* once, which is saved in the file to be used for mapping very large reads.

For a differences of 3, we found 367,946 reads mapped to the reference genome (see Table II). Mapped reads can originate from both and reverse strands, labeled as 0 and 16 respectively in the mapping results. After alignment, we could query for matches/mismatches at different locations (see Table III).

D. Balance Between Running Time And Memory Used

With a difference of 1, we consider runtime and working space with *Suffix Arrays* and *Checkpoint Arrays* differently stored (dependent on parameters c for *Suffix Arrays* and k for *Checkpoint Arrays*) (see Table IV). We observed that, when c and k are 1, the running time is only $\frac{1}{6}$ compared to one when c and k are 100 but the memory requirement is double. When c and k are equal to 30 and 60 respectively, the run time and memory required are not much different from the case of using the minimum *Suffix Arrays* and *Checkpoint Arrays*.

TABLE III
THE SEQUENCE ALIGNMENT RESULTS WITH THREE DIFFERENCES

Name	Direction	Location	Alignment
100062/2	16	13 030	0C0C0A248
100104/1	0	14 275	10G68T52T50
100160/1	0	9679	1C0A0C88
100160/2	16	9679	1C0A0C88
100223/2	16	14 314	0C67T24T157
100269/1	0	14 358	49T51G55G67
10027/2	16	13 128	0C0A24C224
100473/1	0	14 219	31A26A64C53
1005/1	0	17 162	57A78C35T12
100727/2	16	14 477	66A8C27A2

On highly configurable computing systems, e.g. 196 vCPU and 3.75 TB of memory on the Google Cloud platform, reducing memory requirements might not be the most important

TABLE IV
APPROXIMATE MATCHING ALGORITHM WITH DIFFERENT PARAMETERS

	c, k			
	1	*30*	*60*	*100*
Time (s)	234.31	1337.09	1361.23	1393.58
Memory (MiB)	270	274	254	131

concern, while shorting the computing time is more interest. However, in laboratory settings where computing resources could be limited, working with big data would require a solution for the economic problem involving the balance between the two criteria.

V. DISCUSSION

In this paper, we have successfully applied Golang to solve the problem of multiple approximate pattern matching with the concurrency technique. We have designed a buffered channel as a part of the goroutine initialization step. What stands out is that we do not need to close the channel, neither can we close it while goroutines are still sending data to it. No errors, however, arised even though the data in the channel is constantly changes. These channels help synchronize goroutine communication between different work phases and produce consistent results with all runs. This also means that there is no error when goroutines receive and send data across channels. On the other hand, the work of goroutines is in the correct order even though goroutines may be nested within each other and the process of working concurrently is complicated.

Last but not least, our algorithm allows a balance between computation time and memory used making it applicable to different scenarios. When the workload is moderate, achieving the highest computation speed would be of less consideration. Popular online platforms such as Galaxy (https://usegalaxy.org/) can easily meet the desired performance for analysis of small to medium sequence data. In practice, however, data is usually generated in a vast amounts, hence using a system that can receive, send, store, and process big data quickly is essential. Our algorithm allows the parameters to be adjusted so that it does not exceed the maximum limit on system memory and ensures the fastest possible computation time during the implementation.

REFERENCES

[1] Shiraishi, Yuichi, Yusuke Sato, Kenichi Chiba, Yusuke Okuno, Yasunobu Nagata, Kenichi Yoshida, Norio Shiba, et al. "An empirical Bayesian framework for somatic mutation detection from cancer genome sequencing data," Nucleic Acids Res, vol. 41, no. 7, pp. e89–e89, Apr. 2013, doi: 10.1093/nar/gkt126.

[2] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," Bioinformatics, vol. 25, no. 14, pp. 1754–1760, Jul. 2009, doi: 10.1093/bioinformatics/btp324.

[3] Burrows, M., and D. J. Wheeler. "A Block-Sorting Lossless Data Compression Algorithm," 1994.

[4] U. Manber and G. Myers, "Suffix Arrays: A New Method for On-Line String Searches," SIAM J. Comput., vol. 22, no. 5, pp. 935–948, Oct. 1993, doi: 10.1137/0222058.

[5] Compeau, Phillip. Bioinformatics Algorithms: An Active Learning Approach by Phillip Compeau, Pavel Pevzner (2014) Paperback. La Jolla, CA: Active Learning Publishers, 2014.

[6] A. A. A. Donovan, Go Programming Language, The, 1 edition. New York: Addison-Wesley Professional, 2015.

[7] G. Fisher and C. Yeh, "Comparing Producer-Consumer Implementations in Go, Rust, and C," p. 7.

[8] Md. Vasimuddin, S. Misra, H. Li, and S. Aluru, "Efficient Architecture-Aware Acceleration of BWA-MEM for Multicore Systems," in 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May 2019, pp. 314–324, doi: 10.1109/IPDPS.2019.00041.

[9] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu, "A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays," Algorithmica, vol. 48, no. 1, pp. 23–36, May 2007, doi: 10.1007/s00453-006-1228-8.

[10] D. Okanohara and K. Sadakane, "A Linear-Time Burrows-Wheeler Transform Using Induced Sorting," in String Processing and Information Retrieval, Berlin, Heidelberg, 2009, pp. 90–101, doi: 10.1007/978-3-642-03784-9-9.

[11] P. Ferragina and G. Manzini, "Indexing compressed text," J. ACM, vol. 52, no. 4, pp. 552–581, Jul. 2005, doi: 10.1145/1082036.1082039.

[12] Yuki, Koichi, Miho Fujiogi, and Sophia Koutsogiannaki. "COVID-19 Pathophysiology: A Review." Clinical Immunology (Orlando, Fla.) 215 (June 2020): 108427. https://doi.org/10.1016/j.clim.2020.108427.

[13] KwaZulu-Natal Research Innovation and Sequencing Platform (UKZN). "KPCOVID_0471 - KRISP Severe acute respiratory syndrome coronavirus 2 (SARS-CoV-2) virus sequencing in South-Africa... - SRA - NCBI." Accessed August 19, 2020. https://www.ncbi.nlm.nih.gov/sra/SRX8838358.

[14] Blankenberg, Daniel, Assaf Gordon, Gregory Von Kuster, Nathan Coraor, James Taylor, and Anton Nekrutenko. "Manipulation of FASTQ Data with Galaxy." Bioinformatics 26, no. 14 (July 15, 2010): 1783–85. https://doi.org/10.1093/bioinformatics/btq281.

[15] Wu, Fan, Su Zhao, Bin Yu, Yan-Mei Chen, Wen Wang, Zhi-Gang Song, Yi Hu, et al. "A New Coronavirus Associated with Human Respiratory Disease in China." Nature 579, no. 7798 (March 2020): 265–69. https://doi.org/10.1038/s41586-020-2008-3.