Implementation of hyyrö's bit-vector algorithm using advanced vector extensions 2

Kyle Matthew Chan Chua [a,1,*], Janz Aeinstein Fauni Villamayor [a,2], Lorenzo Campos Bautista [a,3], Roger Luis Uy [a,4]

[a] Computer Technology Department, College of Computer Studies, De La Salle University, Manila, Philippines
[1] chua.kyle.matthew.c@gmail.com; [2] janzvillamayor@gmail.com; [3] lorauxe99@gmail.com; [4] roger.uy@dlsu.edu.ph
* corresponding author

ARTICLE INFO

Article history
Received May 24, 2019
Revised July 8, 2019
Accepted October 11, 2019
Available online October 29, 2019

Keywords
DNA sequence alignment
Biometrics
Bit-vector algorithm
SIMD computing capabilities
Modern processors

ABSTRACT

The Advanced Vector Extensions 2 (AVX2) instruction set architecture was introduced by Intel's Haswell microarchitecture that features improved processing power, wider vector registers, and a rich instruction set. This study presents an implementation of the Hyyrö's bit-vector algorithm for pairwise Deoxyribonucleic Acid (DNA) sequence alignment that takes advantage of Single-Instruction-Multiple-Data (SIMD) computing capabilities of AVX2 on modern processors. It investigated the effects of the length of the query and reference sequences to the I/O load time, computation time, and memory consumption. The result reveals that the experiment has achieved an I/O load time of $\Theta(n)$, computation time of $\Theta(n^*\lceil m/64 \rceil)$, and memory consumption of $\Theta(n)$. The implementation computed more extended time complexity than the expected $\Theta(n)$ due to instructional and architectural limitations. Nonetheless, it was par with other experiments, in terms of computation time complexity and memory consumption.

1. Introduction

The Deoxyribonucleic Acid (DNA) is a complex molecule that contains hereditary and biological information which is found in every organism [1]. A DNA sequence can be up to 3 billion in length and is composed of nucleotide bases, namely, Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). Each nitrogenous base holds genetic information and its arrangement in a genome dictates the unique genetic characteristics possessed by a living being. However, researchers discovered that the DNA sequences of all humans are nearly identical; thus, locating and analyzing the similarities or differences would yield more profound knowledge on the function or relationship between the sequences [2][3]. Understanding the sequence's structure and function has made significant impacts on scientific, biological, and medical advancements [4]. Bioinformatics is the science that applying computer science and mathematics to create computational techniques for the collection and analysis of biological data [3]. One of the major researches in the field is performing pattern matching between DNA sequences which leads to the discovery and understanding of biological relationships. It can be used in higher-level processes, such as phylogenetic trees, genetic structure prediction, and disease diagnosis [5][6].

Given a reference sequence length n and a query sequence with length m, and the goal of sequence alignment is to compute the edit distance (score) between the sequences. Then, most of the time determines within the pre-defined k-error thresholds to pinpoint regions of similarities that allow the analysis and assessment of relationship between species and organisms [5]–[8]. The reference is the

source sequence (e.g. a human genome), which can be obtained from public online GenBank sequence database, usually from NCBI [9]. Whereas, the query sequence is a short read that the scientists are interested in locating or investigating (e.g. genomic mutation or diseases) from the reference string. Edit distance could be defined as the number of required edit operations to make both sequences equal. Moreover, it can concurrently process depending on the number of sequences. Sequence alignment algorithms can be classified in either pairwise or multiple sequence alignment; the former aligns exactly 2 sequences (one query, one reference), while the latter aligns 2 or more sequences simultaneously [6]. Researchers argue that multiple sequence alignment is more significant for scientific and research purposes. However, it is essential to note that multiple sequence alignment is merely an extension of pairwise sequence alignment. Thus, multiple sequence alignment benefits from enhancing pairwise sequence alignment [5]. Sequence alignment algorithms can be divided into two types; Global or Local. The global method aligns the sequences from end-to-end, and it is useful when identifying the total similarity of sequences. While, the local method aligns fragments of the sequences, and it is useful when identifying homologous regions [10].

DNA sequence alignment is a computational-heavy and time-demanding process because of its time complexity usually dependent on both m and n [8]. Due to the advancements on DNA Next Generation Sequencing (NGS) technologies, scientists were able to generate DNA sequences at a much higher rate and lower cost, and DNA sequence alignment could not keep up with the rapid growth of sequence database; therefore, There are challenged to formulate efficient bioinformatics solutions which can be crucial in numerous scenarios, such as DNA forensics, early diagnosis of susceptibility to genetic diseases, and prevention of bacteria or virus evolution [5]. This influenced the implementation of DNA sequence alignment solutions on High-Performance Computing (HPC) technologies that can perform intensive computational processes, including Field Programmable Gate Arrays (FPGA); Graphical Processing Units (GPU); and Cell Broadband Engines (Cell BE) [11]. However, as of today, few research works has been conducted that focuses on implementing sequence alignment using Intel's iinstruction-set architecture to run on General Purpose Processors (GPP).

In 2013, Intel introduced the Haswell microarchitecture, which featured Single-Instruction-Multiple-Data (SIMD) capabilities as it supported Advanced Vector Extensions (256-bit operators), an extension from Streaming SIMD Extensions (128-bit operators) [12]. These instructions exploit the data stream's parallelism allowing it to process multiple data simultaneously with a single instruction improving the throughput of floating-point operations [13][14]. The addition of SIMD instructions to Intel processors offers a rich instruction set, making it possible to implement a DNA sequence alignment algorithm to run on GPP.

In this study, the researchers implemented an existing bit-vector algorithm that performs DNA sequence alignment on a query sequence and a reference sequence. This study took advantage of modern processors' bit-parallel operation capabilities utilizing Intel's SIMD technologies, specifically, Advanced Vector Extension 2 (AVX2), supported by at least 4th generation Intel processors (code-named "Haswell"). The correctness of the program was verified through multiple test cases. Furthermore, this paper also highlights the program's performance with various DNA sequences by measuring execution time and memory consumption. The study mainly focused on implementing Hyyrö's bit-vector algorithm [7] to utilize AVX2 instruction set architecture for pairwise sequence alignment. The system would be capable of handling up to 256 query sequence length since the query sequence and the bit-vector variables were processed in the 256-bit vector registers. Real-world DNA sequences obtained from the National Center for Biological Information (NCBI) online GenBank sequence database [9] utilized as a data set for experimentation.

2. Method

2.1. Sequence Alignment Algorithms

Several sequence alignment algorithms have been developed based on the dynamic programming approach; most notable are the Needleman-Wunsch and Smith-Waterman algorithm [11]. Both

algorithms are useful primarily for pairwise and global alignment. The advantage of using the Needleman-Wunsch and Smith-Waterman algorithm is the capability to locate the optimal alignment between the sequences. However, these algorithms demand more time to complete and run at $\Theta(nm)$ [15][16]. Shehab *et al.* [15] developed the FDASA (Fast Dynamic Algorithm for Sequence Alignment) which executes the Needleman-Wunsch and Smith-Waterman algorithm with faster time complexity of either $\Theta(3m+1)$ when two sequences have equal length ($\Theta(3m+2)$) or different lengths. Tarhio and Ukkonen [17] unveiled that the Boyer-Moore algorithm-generated optimal runtime speed for longer sequences, though increasing the k mismatch threshold will slow down the computation compared to other dynamic programming algorithms. Having said that, Gou [18] highlighted the difference between the Naïve, Knuth-Morris-Pratt, Boyer-Moore, and Rabin-Karp algorithm in terms of alignment speed for various sequence lengths. The results supported Tarhio and Ukkonen's [17] argument that the Boyer-Moore algorithm works best for longer sequences. On the other hand, it was revealed the Rabin-Karp algorithm is suitable for shorter sequences. Other researchers have delved into finite state machines to develop sequence alignment algorithms. For instance, the Aho-Corasick algorithm is one of the most commonly used algorithms that use an automata approach for exact multiple string matching. Subsequently, the Commentz-Walter algorithm was introduced as a better alternative for the Aho-Corasick algorithm since it is a combination of both Aho-Corasick and Boyer-Moore algorithm [19]. In a comparative study by Vidanagamachchi *et al.* [19], the results invalidated prior belief because the Aho-Corasick algorithm attained better runtime than the Commentz-Walter algorithm because the latter requires more pre-processing time to construct the finite state machine. Zhu *et al.* [20] formulated the Bayes block aligner algorithm for local alignment that incorporates the statistics concept of Bayes inference, which involves probability and distribution, to mitigate the need of defining parameters and variables, such as gap penalties and scoring matrices [21]. The study shows that the Bayes block aligner algorithm outperformed the widely known SSEARCH algorithm on VAST in terms of the percentage of correctly identifying structural neighbors while achieving a time complexity of $\Theta(n^2)$ [20].

Aside from the algorithm, the edit distance metric also plays an important role in sequence alignment performance. Pandiselvam *et al.* [16] conveyed that the simplest edit distance to compute is the Hamming distance because it merely counts the number of differences at every position between sequences with equal length. The Hamming distance is mainly used for exact sequence alignment since it requires the sequences to have the same length and it only performs substitution operation. Another study from Levenshtein [22] explored the use of binary information in which mismatches can be corrected using deletions, insertions, and substitutions. The scoring scheme is called the Levenshtein distance; this metric is used for approximate sequence alignment because it is not constrained by the length of the sequences and offers more edit operations. It follows a dynamic programming approach that counts the minimum cost that is required for two sequences be equal. Research contrasted the two edit distance metrics and the investigation has proven that although the Hamming distance generated more accurate alignment results, the Levenshtein distance proved to be faster by achieving $\Theta(n+m)$ time complexity compared to the former's $\Theta(nm)$ time complexity [16].

A number of researchers have implemented sequence alignment algorithms by utilizing the computing capabilities of the SIMD unit embedded in GPPs since it is much easier to program, more portable, and widely available [11]. Nataliani and Wellem [23] implemented Myer's bit-vector algorithm using MATLAB to investigate the similarity of Rhodopsin protein sequence of class Aves. To conduct the experimentation, the data set consists of the sequences of 25 species that have Rhodopsin protein from class Aves that were obtained from the Universal Protein Resource (UniProt) Consortium website and DNA Data Bank of Japan (DDBJ) website. The study mainly features a proof-of-concept implementation of the bit-vector algorithm using a high-level tool. However, it falls short of evaluating the speed and memory performance of the application.

Fredriksson [24] featured an alternative method to perform string matching using Myers' bit-parallel algorithm. The researcher proposed a new arrangement for comparing short query sequences ($m < w$, where w is the computer word size) such that the computations are performed in a row-wise approach

instead of a column-wise manner to minimize the wasted bits of the computer word. The algorithm was implemented on an Intel Pentium 4 processor, coded using Intel SSE2 instruction set architecture through C/C++ intrinsics. For experimentation, the researchers used a randomly generated DNA sequence of size 64Mb as reference sequence and short query sequences with varying lengths (i.e. 8, 16, 32, 64, 128) to investigate the effects of varying w. The results showed that the execution time of the whole sequence alignment process has a linear relationship with m, and subsequently, w. The researchers argued that their implementation is very fast, however, it is dependent on the architecture.

Faro and Külekci [25] promoted an exact string-matching method, called Exact Packed String-Matching algorithm (EPSM), which aims to speed up the process for short query sequences. The idea is to exploit the bit-parallelism of the word RAM model; thus, the computations are performed on words of length w (assuming w is 32). The researchers utilized Intel SSE's specialized packed string matching intrinsics that includes: *wscmp*, *wsmatch*, *wsblend*, and *wscrc*. To evaluate the performance of the proposed algorithm, the reference sequences used were a genome sequence, a protein sequence, and an English natural language text, all of which are 4Mb in size; moreover, sets of 1000 query sequences were extracted from each corresponding reference sequence, where m would range from 2 to 32. The results revealed that their implementation has achieved a worst case of $O(nm)$ time complexity and $O(2^k)$ memory consumption. Comparing it with other algorithms, the researchers argued that the EPSM algorithm is the fastest when $m \leq 32$.

Memeti and Pllana [6] presented a large-scale DNA analysis algorithm designed to be implemented on the Intel Xeon Phi 7120P coprocessor (code-named "Knights Corner"). The proposed algorithm was based on finite automata, it exploits thread-level parallelism by dividing and distributing the input DNA sequence across threads; moreover, it also takes advantage of bit-parallelism featured in AVX-512 instruction set architecture. The DNA sequences of mouse, cat, dog, chicken, human, and turkey obtained from the GenBank sequence database of NCBI composed the reference sequence data set, while *regex-dna* benchmark with a fixed number of errors composed the query sequence data set for evaluation. Each test case was executed 20 times to prove the consistency of its performance. The results reported a maximum speedup of 10x compared to a sequential implementation on the Intel Xeon ES-2695v2 processor. The researchers were interested to investigate the optimal number of threads for multiple sequence alignment. In contrast, since our research work focuses on pairwise sequence alignment, this approach is not applicable to our study.

2.2. Sequence Alignment through Bit-vector Algorithm

The prevailing method for aligning two sequences is via the dynamic programming method. Dynamic programming incorporates a recursive approach which usually requires an $(m+1)(n+1)$ two-dimensional scoring matrix. However, the run time of the algorithms using this approach is highly dependent on both m and n, and sometimes even k-error threshold, and consumes $\Theta(mn)$ space [26].

Myers [8] proposed an alternative solution in finding the local alignment between a query and a reference to solve for the Levenshtein distance, a sequence alignment metric that allows 3 edit operations, namely, insertion, deletion, and substitution [7]. Myers's algorithm, widely known as Myers bit-vector algorithm, follows a dynamic programming approach that takes advantage of bit-parallel operations featured in modern processors [27]. It assumes a register size of 32 or 64, therefore restricting the length of the query sequence to the word size w [8]. Generally, the approach of the algorithm is to solve the matrix in columns rather than computing each cell individually. Each column is encoded using m-bits vector representation, namely, *Pv* for the positive vertical delta value, *Mv* for the negative vertical delta value, *Ph* for the positive horizontal delta value, *Mh* for the negative horizontal delta value, *Xv* for the current vertical column value, and *Xh* for the horizontal column value. This also follows an observation lemma that the difference between the adjacent values in each cell in the matrix has a value of either -1, 0, or +1. The matrix is completely solved once it has iterated through the whole reference sequence. Therefore, the algorithm can achieve a runtime of $\Theta(n)$ assuming that operations will execute at $\Theta(1)$, which is promoted to be the fastest sequence alignment algorithm as of now [27]. Hyyrö [7] modified Myers' [8] bit-vector algorithm to compute for the Damerau-Levenshtein distance between a query and a reference. The Damerau-Levenshtein distance extends the Levenshtein distance by

including transposition between two adjacent characters, therefore, allowing a total of 4 edit operations [7]. The addition of transposition edit operation is achieved through the vector variable *Xp*. The algorithm consists of bit operations, namely, | (OR), & (AND), ^ (XOR), << (left shift), + (bitwise addition), including arithmetic and comparison operations [26].

The algorithm requires a pre-processing of the query sequence. It involves translating each character from the query into its corresponding bit-mask that represents its position in the text. The index in the vector will be set to 1 when the corresponding character occurs in the query at the specific index, and 0 otherwise. For example, the bitmask of character 'A' for the query "ACTGAC" is B['A'] = b'100010 [28].

2.3. Advanced Vector Extension 2 Instruction Set Architecture

The SIMD computing capabilities featured in GPPs enabled vector operations to be executed within a single clock cycle [29]. In efforts to expand the Streaming SIMD Extensions (SSE) computing technology, Intel released the Advanced Vector Extensions (AVX) and AVX2 featured in the Sandy Bridge microarchitecture and Haswell microarchitecture respectively [30]. The AVX and AVX2 extend the SSE single-precision floating-point, double-precision floating-point, and integer commands to operate on 256-bits YMM vector registers while also increasing the peak double-precision ops per cycle [31]. Legacy SSE instructions can still be utilized to execute on the lower 128-bits of the YMM registers, this provides access to one of the key features of SSE, text string processing instructions. These instructions aim to speed up a number of string primitives whose process would usually entail non-optimal utilization of the processor and its instruction pipelines. In addition, the Vector Extension (VEX) prefix instruction encoding format was introduced, enabling three-operand syntax, in some cases four-operand, using non-destructive source operands [32]. Although the AVX2 instruction set architecture offers a substantial amount of floating-point and integer instructions, it is not capable of performing 256-bit arithmetic addition and bit shift. Thus, the researchers must develop simulations of these operations to satisfy the requirements of the bit-vector algorithm.

2.4. Research Design

This study provides a discussion on the implementation of a bit-vector algorithm using AVX2 instruction set architecture as well as its performance evaluation with real-world DNA sequences. Fig. 1 presents the algorithm used for this study, it was developed and presented by Hyyrö [7] in his own paper; Hyyrö did not present any performance evaluation since his study focused on the theories and framework of the algorithm. For the purposes of this study, the algorithm was modified (See lines 14 and 15 on Fig. 1) such that the computation for the Damerau-Levenshtein distance will continue regardless of when the *k*-error threshold has been reached. This not only enables the evaluation of similarity between the two sequences but also allows pinpointing highly similar regions. The pre-processing of the query sequence was also modified to obtain the reverse bitmask of each character.

The algorithm was implemented on the Visual Studio 2017 and compiled with Microsoft Macro Assembler. The application is composed of 2 elements: the C++ program and the assembly program. The former handles the input and output (I/O) of the application which is interfaced with the latter that is responsible for computing the Damerau-Levenshtein distance between the query and the reference sequences. Initially, the application reads the text files that contain the query and reference sequences through a memory mapping method that involves allotting a chunk of memory space where the lengthy sequences will be placed in by the operating system and stores them in their corresponding string variable. The length of the query string will be determined which will be passed along with the addresses of the query and reference strings as arguments whenever the assembly program is invoked. The assembly program uses a flat memory model and C calling convention. The bit-vector variables of the algorithm are loaded into the YMM registers from memory whenever it is used for calculation allowing up to 256 query sequence length. The implementation requires the data to be shifted to the most significant bit of the register, like zero-extending, to avoid tampering of the higher-order bits during calculation which will affect the result.

```
1  <Preprocess B[σ] with P>                          ;Preprocess of bit-vectors for sequence P
2  Bit-vector Pv,Mv,Ph,Mh,Xv,Xh,Eq,Xp                ;Setup vectors with 0m
3  Score = m
4  Pv = 1^m
5  Mv = 0^m                                          ;Initialize vertical delta values
6  for j = 1, 2, ..., n do
7       Eq = PEq[Σ[T[j]]]                            ;Bit-vectors for reference symbol j
8       Xv = Eq | Mv
9       Xh = (((-Xh) & Xv) << 1) & Xp
10      Xh = Xh | (((Xv & Pv) + Pv) ^ Pv) | Xv | Mv  ;compute current delta vector
11      Ph = Mv | ~ (Xh | Pv)
12      Mh = Xh & Pv                                 ;update horizontal delta values
13      Xp = Xv                                      ;store old pattern bit-vector
14      if(Ph & 10m-1) then score += 1
15      else if(Mh & 10m-1) then score -= 1          ;update score
16      Xv = (Ph << 1)
17      Pv = (Mh << 1) | ~ (Xh | Xv)
18      Mv = Xh & Xv                                 ;update vertical delta values
```

Fig. 1. Hyyrö's [7] bit-vector algorithm for computing Damerau-Levenshtein distance.

The flowchart for pre-processing the query sequence for a character is shown in Fig. 2. It utilizes a series of *vpcmpistrm* instructions to obtain the reverse bitmask of a character. The *vpcmpistrm* instruction can process at most 16 characters (resulting to 16 bits of the bitmask) at a time. Thus, requiring a total of $\lceil m/16 \rceil$ to obtain the whole bitmask of the query sequence. The upper half and the lower half of the bitmask must be obtained separately since they are processed in the 128-bit XMM registers. After looping through the whole query sequence, the upper and lower bitmasks are merged through the *vperm2i128* instruction. Since the order of the word elements in the YMM vector register is reversed, the *vpshufb* instruction is utilized to shuffle the position of the word elements and accurately reflect the query sequence. It also follows that the data should be on the most significant bit of the register. The pre-processing stage is executed for characters 'A', 'C', 'G', and 'T'.

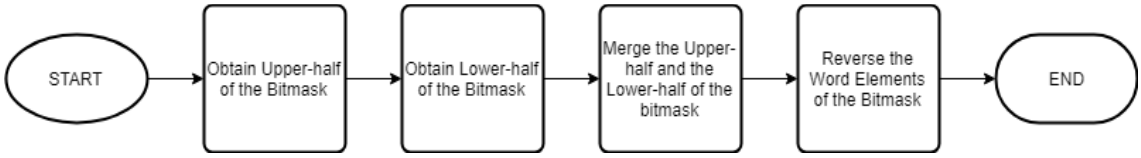

Fig. 2. Flowchart for obtaining the bitmask of a character in the query sequence.

The AVX2 provides a rich set of instructions allowing for a fairly straightforward implementation of the bit-vector algorithm. The | (OR) operation corresponds to the *vorps* instruction, the & (AND) operation corresponds to the *vandps* instruction, the ^ (XOR) operation corresponds to the *vxorps* instruction, and the ~ (NOT) operation can be performed simply by performing an XOR to the argument and all ones. However, the SIMD instruction set architecture does not support 256-bit wide addition and left shift because the vector elements are treated independently during calculation (i.e. No carry between vector elements). Thus, the researchers must simulate these two instructions.

A combination of store, load, and 32-bit addition were utilized to perform 256-bit wide addition. Initially, the two 256-bit arguments are stored in memory and the carry flag is cleared. The arguments are treated as 32-bit chunks by loading them into the 32-bit general-purpose registers and added by executing *adc* instruction. This replicates the addition and carry-over between doubleword elements of the vector register. The process is repeated 8 times to accomplish 256-bit wide addition.

Simulating the 256-bit wide left shift involves storing a copy of the argument prior to executing *vpsllq* instruction which will shift the quadword elements of the YMM register to the left by 1 bit. This would allow the retrieval of the most significant bit of each quadword element that would have been lost

after performing the shift instruction. To reproduce the carryover, the most significant bit of every quadword element (i.e. bit 63, 127, 191) is checked to see if it is set, if so, the next quadword element is incremented, otherwise, no action is taken.

After each iteration, the assembly program calls a C++ function passing the calculated score at the current index as parameter to check whether it is equal with the current lowest score, if so, the specific index is added into the array of indexes with the same score, if it is otherwise lower than the current lowest score, the previous score and the list of indexes is overwritten, otherwise, no action is taken. The application outputs the results of the computation by writing the summary in the console and text file. The summary includes the query string, length of the query, length of the reference, lowest score, and possible substrings where the lowest score may be located.

Additionally, the implementation follows some optimization guidelines from the Intel® 64 and IA-32 Architectures Optimization Reference Manual [33] that includes: keeping code and data on separate pages, aligning data on natural operand size address boundaries, using test instruction instead of *cmp* whenever possible, using *add* or *sub* instructions instead of *inc* or *dec*, using logical instructions to zero a register, unrolling loops, arranging code to be consistent with the static branch prediction algorithm or to reduce branches, utilizing single-precision instructions instead of double-precision, taking advantage of zero-latency *mov*, organizing code to maximize micro-architectural resources, and enabling flush-to-zero and denormals-are-zero mode.

To evaluate the performance of the study's implementation, the DNA sequences of *Homo sapiens* (human), *Mus Musculus* (mouse), *Solanum Pennellii* (eudicots), *Brachypodium Distachyon* strain Bd21 (stiff brome), *Ornithorhynchus Anatinus* (platypus), *Cajanus Cajan* (pigeon pea), *Pseudomonas Syringae* (g-proteobacteria), *Chthonomonas Calidirosea* (bacteria), *Prochlorococcus Marinus str. MIT 9211* (cyanobacteria), and *Mycoplasma Conjuctivae* (mycoplasmas) were selected for experimentation. The reference sequence dataset is composed of chromosome 1 sequences from the chosen species which can be obtained from the GenBank sequence database of NCBI [9]. For this study, the researchers have omitted the instances of the wildcard character 'N' for all sequences. Table 1 shows the reference sequence datasets and their corresponding length, excluding character 'N'. On the other hand, the query sequence dataset is composed of generated DNA sequences that have varying lengths of 32, 64, 92, 128, 160, 192, 224, and 256.

Table 1. Summary of dataset used.

Species	Reference Genome	
	Assembly Name	*Length (No. of Characters)*
Human	GRCh38.p12	230481014
Mouse	GRCm38.p4 C57BL/6J	195471971
Eudicots	SPENNV200	109333515
Stiff Brome	Bd21	75071545
Platypus	Ornithorhynchus_anatinus-5.0.1	47594283
Pigeon Pea	C.cajan_V1.0	17676265
G-proteobacteria	DC3000	6397126
Bacteria	T49	3437861
Cyanobacteria	MIT 9211	1688963
Mycoplasmas	HRC/581T	846214

3. Results and Discussion

The experiment aims to investigate the effect of diverse sequence lengths on I/O load time, computation time, process memory, and power consumption of the implementation. The procedure was performed on the Dell XPS 15 laptop equipped with Intel Core ™ i7 –6700HQ 2.6Ghz 64-bit processor and 8GB of RAM. Each test case is executed 10 times to average out the result.

To validate the correctness of the implementations, the application was tested to return the score of aligning a query sequence against 3 variations of the same sequence: no mismatch, with 5 mismatches,

and with the random number of mismatches. The score was expected to correspond to the number of mismatches the researchers inserted. Moreover, the application was cross-validated with the Python implementation of the algorithm using Mycoplasmas' sequence as the reference against the query sequence dataset. The results are consistent with the expected scores, thus, verifying the correctness of the implementation.

For the purpose of proving that the implementation was optimized to enhance computation time, the performance of the optimized implementation was compared against the barebone implementation (i.e. no optimizations done). Fig. 3 to Fig. 6 show the comparison of the average computation times between the two versions. Based on the data, the improvement is apparent as the optimized version outperformed the barebone version by achieving a speedup of up to 1.36 times.

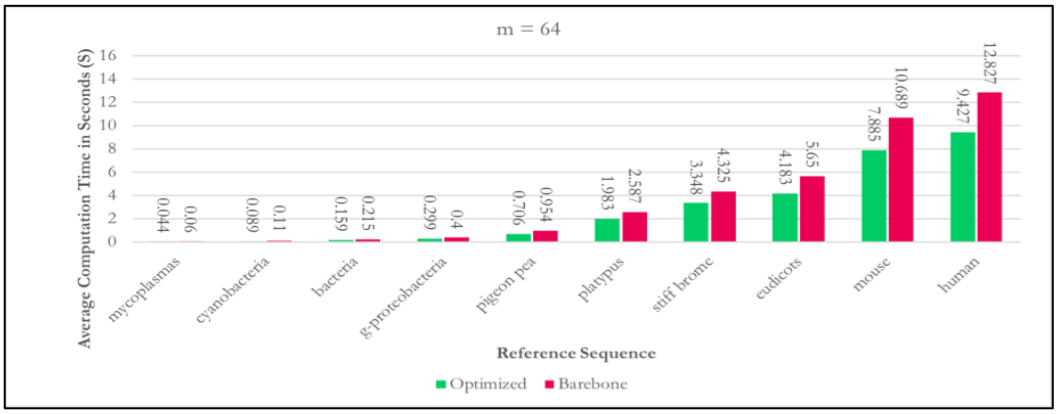

Fig. 3. Comparison of average computation time in seconds (S) using query sequence with length of 64.

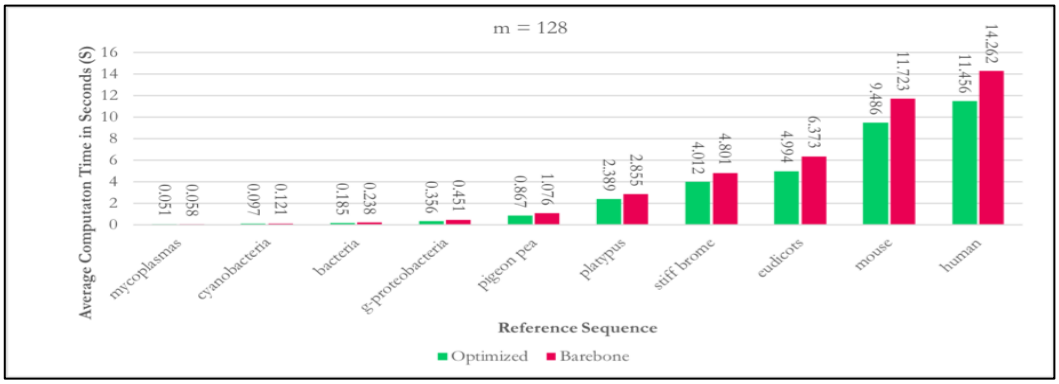

Fig. 4. Comparison of average computation time in seconds (S) using query sequence with length of 128.

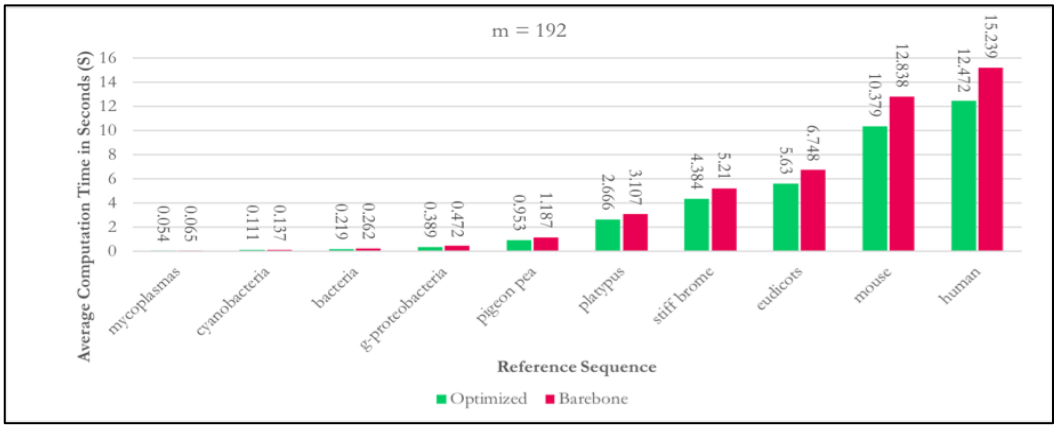

Fig. 5. Comparison of average computation time in seconds (S) using query sequence with length of 192.

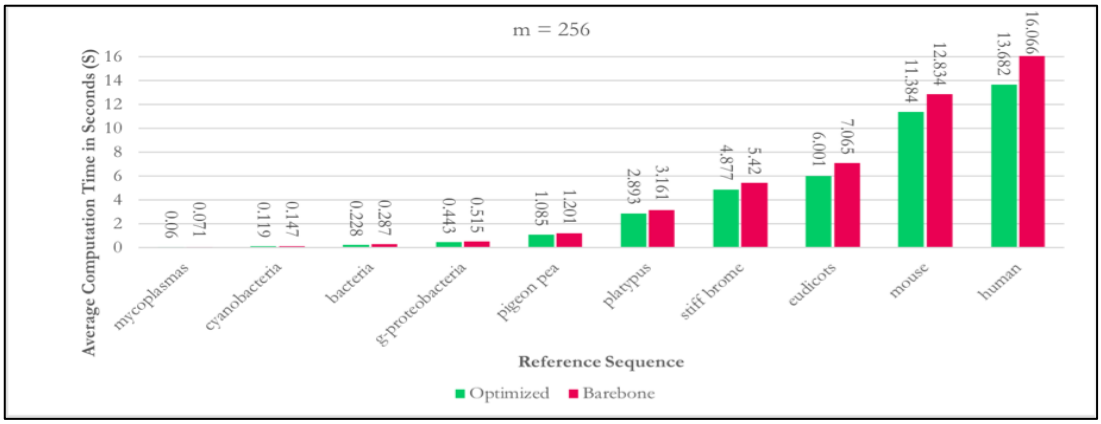

Fig. 6. Comparison of average computation time in seconds (S) using query sequence with length of 256.

The average I/O load time is illustrated in Fig. 7. The I/O load time measures how long it takes to read the query and reference sequences text files and store it in their corresponding memory space. It is evident that the length of the reference sequence has a linear effect to the I/O load time, while the length of the query sequence has little to no impact; thus, its time complexity is $\Theta(n)$.

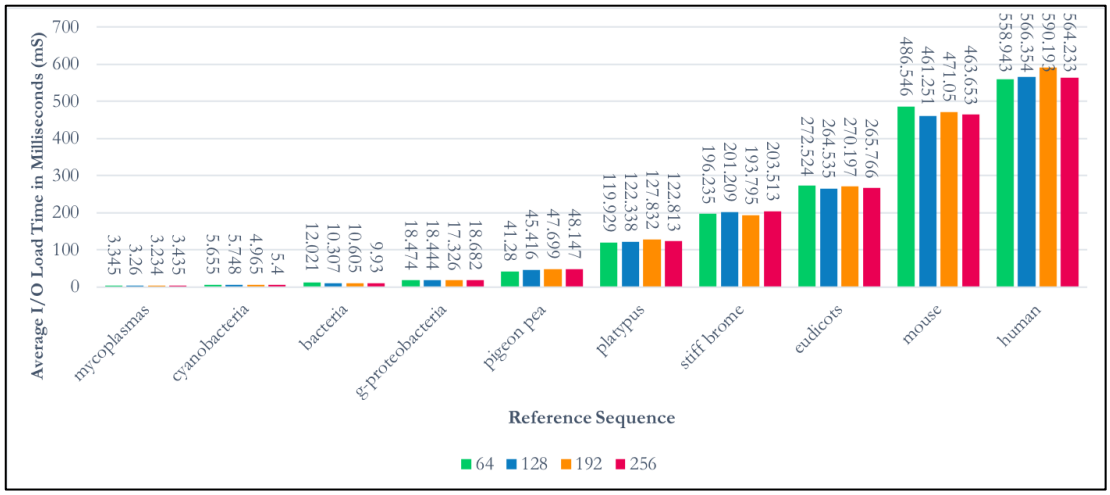

Fig. 7. Average I/O load time in milliseconds (mS).

The computation time consists of a pre-processing stage, actual computation for Damerau-Levenshtein distance, and storing of indexes where the most similar substrings may be located. Fig. 8 shows the performance of the application in terms of computation time. It can be derived that it is heavily dependent on the size of the reference sequence. However, the data also shows that the length of the query sequence affects the computation time which rejects the expected time complexity of $\Theta(n)$. Despite limiting the size of the query ($m < 256$) and performing the calculations on the 256-bit YMM registers, the influence of the length of the query to the computation time is caused by the accumulation of some frequently repeated process that is dependent on m. For example, the simulated 256-bit wide arithmetic addition is completed faster for shorter query sequences because the carry out is cascaded less. Further investigation (Shown in Fig. 9 and Fig. 10) reveals that the machine word size is 64 bits since most of the AVX/AVX2 instructions used in the implementation operates by quadwords (64 bits). Therefore, since the computation time displays a linear relationship with the reference sequence size and is also affected by the length of the query sequence and machine word size, it runs at $\Theta(n*\lceil m/64 \rceil)$. Moreover, Fredriksson [24] had a similar finding in his study that supports this hypothesis wherein the researcher attributed it to be caused by hardware limitation such that the SIMD instructions still execute based on the native machine word size (denoted as w); therefore the bit-operations do not run at the expected $\Theta(1)$, but rather $\Theta(\lceil m/w \rceil)$.

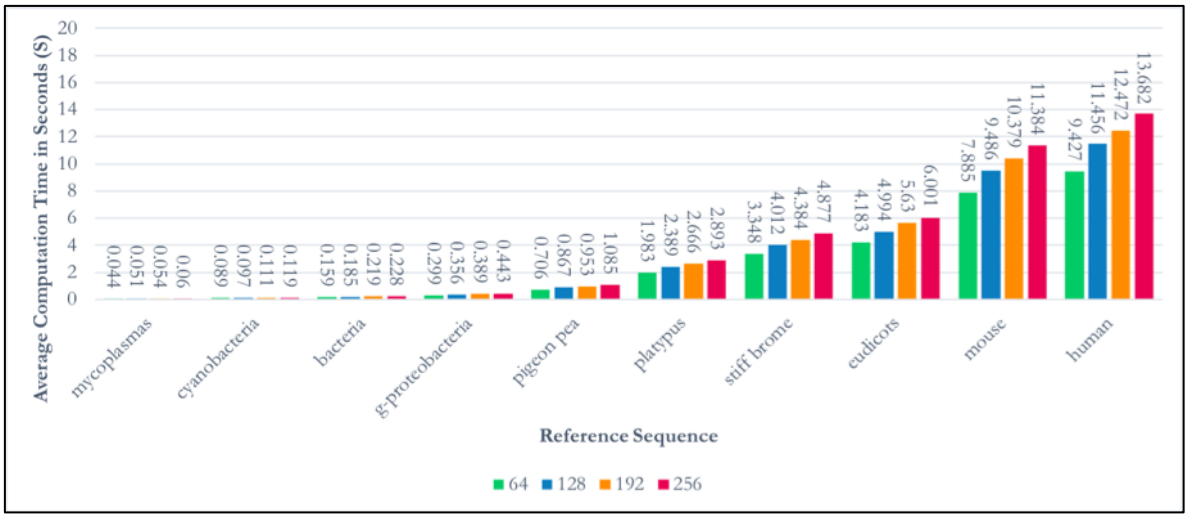

Fig. 8. Average computation time in seconds (S).

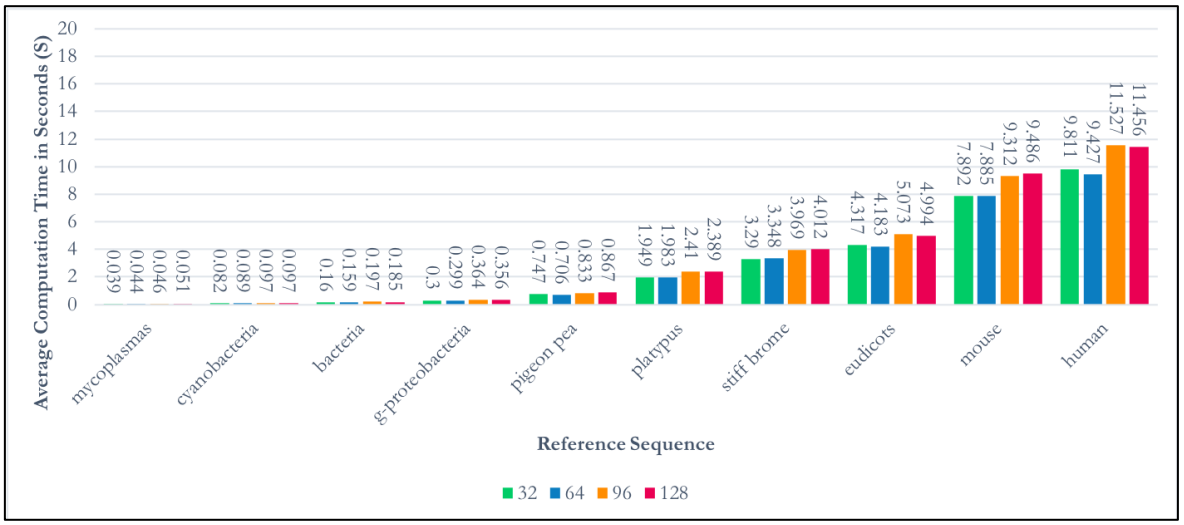

Fig. 9. Average computation time in seconds (S) using query sequences with lengths of 32, 64, 96, 128.

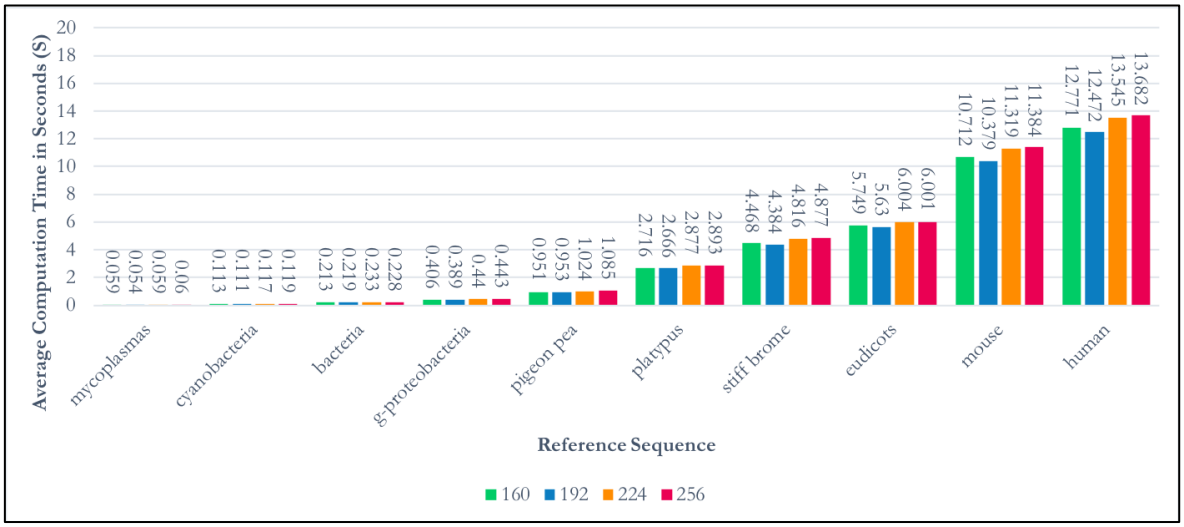

Fig. 10. Average computation time in seconds (S) using query sequences with lengths of 160, 192, 224, 256.

Fig. 11 shows the memory consumption of the application in megabytes (MB). A linear relationship between the memory consumption and reference sequence length can be deduced from the illustration.

The memory consumption was consistent for each reference data-set regardless of the length of the query sequence. It consumes approximately the size of the reference sequence in bytes plus 40 MB. Therefore, it can be argued that it has achieved $\Theta(n)$ memory consumption. Finally, the power consumption was investigated, and it reveals that the program consumes approximately 20 – 25W regardless of sequence length.

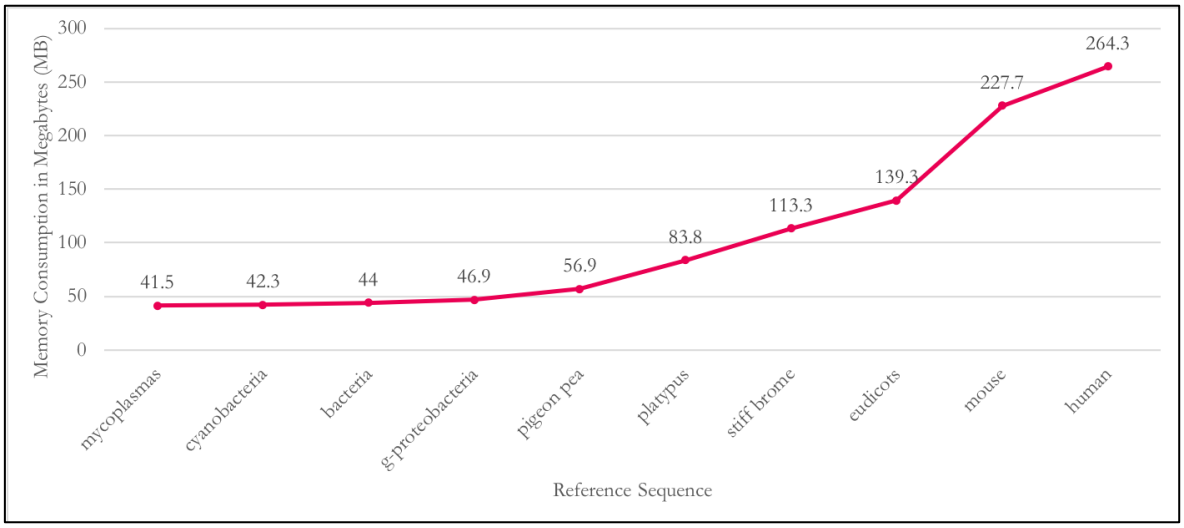

Fig. 11. Memory Consumption in megabytes (MB).

4. Conclusion

This study presents an implementation of Hyyrö's bit-vector algorithm for pairwise DNA sequence alignment using AVX2 instruction set architecture to run on modern processors. To our knowledge, this is an initial attempt of developing the algorithm to take advantage of SIMD computing capabilities of AVX2 on recent processors which advances the idea of the possibility of implementing other compute-intensive applications on GPP. Based on the results of the experimentation, the AVX2 implementation has achieved an I/O load time of $\Theta(n)$ since it is mostly impacted by the length of the reference sequence. It can also be argued that the computation time complexity of the implementation is longer than the ideal $\Theta(n)$ time complexity due to the simulation of the 256-bit addition and left shift which entails the carry out to be cascaded to the higher-order elements, and architectural limitations that causes instruction to operate based on its native machine word size (64-bits) and not on the actual SIMD vector size; thus, each operation runs at $\Theta(\lceil m/64 \rceil)$ and the implementation computes at $\Theta(n^*\lceil m/64 \rceil)$, similar to Fredriksson's [24] implementation and on a par with Faro and Külekci's [25] implementation that reached a computation time of $\Theta(nm)$. Furthermore, the implementation has a memory consumption of $\Theta(n)$, wherein it requires approximately twice the size of the reference sequence in bytes plus 40 MB. This study's implementation displayed a linear growth of memory consumption. In contrast, Faro and Külekci's [25] implementation showed exponential growth. Performing pairwise sequence alignment using Hyyrö's algorithm is just the first step. Future research works may want to attempt extending or removing the limitations on the query sequence length or to explore multiple sequence alignment and multi-core or multi-threaded programming.

References

[1] S. P. Adey, "GPU Accelerated Pattern Matching Algorithm for DNA Sequences to Detect Cancer using CUDA Dissertation," *Coll. Eng. Pune*, 2013, available at : Google Scholar.

[2] A. Mahram, "FPGA acceleration of sequence analysis tools in bioinformatics," *Bost. Univ. Coll. Eng.*, 2013, available at : Google Scholar.

[3] R. Bhukya and D. Somayajulu, "2-Jump DNA Search Multiple Pattern Matching Algorithm," *Int. J. Comput. Sci. Issues*, vol. 8, no. 3, pp. 320–329, 2011, available at : Google Scholar.

[4] I. Murnaghan, "The Importance of DNA," *Explore DNA*, 2019. [Online]. Available: http://www.exploredna.co.uk/the-importance-dna.html.

[5] X. Chang, F. A. Escobar, C. Valderrama, and V. Robert, "Exploring Sequence Alignment Algorithms on FPGA-based Heterogeneous Architectures.," in *IWBBIO*, 2014, pp. 330–341, available at : Google Scholar.

[6] S. Memeti and S. Pllana, "Accelerating DNA Sequence Analysis Using Intel(R) Xeon Phi(TM)," in *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015, pp. 222–227, doi: 10.1109/Trustcom.2015.636.

[7] H. Hyyrö, "A bit-vector algorithm for computing Levenshtein and Damerau edit distances," *Nord. J. Comput.*, vol. 10, no. 1, pp. 29–39, 2003, available at : Google Scholar.

[8] G. Myers, "A fast bit-vector algorithm for approximate string matching based on dynamic programming," *J. ACM*, vol. 46, no. 3, pp. 395–415, May 1999, doi: 10.1145/316542.316550.

[9] "Genome," *NCBI*. [Online]. Available: https://www.ncbi.nlm.nih.gov/genome. [Accessed: 12-Feb-2019].

[10] L. Langner and M. S. D. Weese, "Parallelization of Myers fast bit-vector algorithm using GPGPU," Diploma/Thesis, Freie Universität, Berlin, 2011, available at: Google Scholar .

[11] K. Benkrid, A. Akoglu, C. Ling, Y. Song, Y. Liu, and X. Tian, "High Performance Biological Pairwise Sequence Alignment: FPGA versus GPU versus Cell BE versus GPP," *Int. J. Reconfigurable Comput.*, vol. 2012, pp. 1–15, 2012, doi: 10.1155/2012/752910.

[12] W. Muła, N. Kurz, and D. Lemire, "Faster Population Counts Using AVX2 Instructions," *Comput. J.*, vol. 61, no. 1, pp. 111–120, Jan. 2018, doi: https://doi.org/10.1093/comjnl/bxx046.

[13] "Basics of Single Instruction Multiple Data (SIMD)," *Code Project*, 2011. [Online]. Available: https://www.codeproject.com/Articles/146414/Basics-of-Single-Instruction-Multiple-Data-SIMD. [Accessed: 12-Feb-2019].

[14] C. Lomont, "Introduction to intel advanced vector extensions," *Intel White Pap.*, pp. 1–21, 2011, available at : Google Scholar.

[15] S. A.Shehab, A. Keshk, and H. Mahgoub, "Fast Dynamic Algorithm for Sequence Alignment Based On Bioinformatics," *Int. J. Comput. Appl.*, vol. 37, no. 7, pp. 54–61, Jan. 2012, doi: 10.5120/4624-6636, available at : http://research.ijcaonline.org/volume37/number7/pxc3876636.pdf.

[16] P. Pandiselvam, T. Marimuthu, and R. Lawrance, "A Comparative Study on String Matching Algorithm of Biological Sequences," *CoRR*, vol. abs/1401.7416, 2014, available at : http://arxiv.org/abs/1401.7416.

[17] J. Tarhio and E. Ukkonen, "Approximate Boyer–Moore String Matching," *SIAM J. Comput.*, vol. 22, no. 2, pp. 243–260, Apr. 1993, doi: 10.1137/0222018.

[18] M. Gou, "Algorithms for String matching." July, 2014, available at: Google Scholar.

[19] S. M. Vidanagamachchi, S. D. Dewasurendra, R. G. Ragel, and M. Niranjan, "Commentz-Walter: Any Better than Aho-Corasick for Peptide Identification?," *Int. J. Res. Comput. Sci.*, vol. 2, no. 6, pp. 33–37, Nov. 2012, doi: 10.7815/ijorcs.26.2012.053.

[20] J. Zhu, J. S. Liu, and C. E. Lawrence, "Bayesian adaptive sequence alignment algorithms," *Bioinformatics*, vol. 14, no. 1, pp. 25–39, Feb. 1998, doi: 10.1093/bioinformatics/14.1.25.

[21] B.-J. M. Webb, "BALSA: Bayesian algorithm for local sequence alignment," *Nucleic Acids Res.*, vol. 30, no. 5, pp. 1268–1277, Mar. 2002, doi: 10.1093/nar/30.5.1268.

[22] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, 1966, vol. 10, no. 8, pp. 707–710, available at : Google Scholar.

[23] Y. Nataliani and T. Wellem, "Implementation of Bit-Vector Algorithm for Approximate String Matching on Rhodopsin Protein Sequence," *Int. J. Comput. Appl.*, vol. 72, no. 14, pp. 34–38, Jun. 2013, doi: 10.5120/12565-9214.

[24] K. Fredriksson, "Row-wise Tiling for the Myers' Bit-Parallel Approximate String Matching Algorithm," 2003, pp. 66–79, doi: 10.1007/978-3-540-39984-1_6.

[25] S. Faro and M. O. Külekci, "Fast Packed String Matching for Short Patterns," 2013, pp. 113–121, doi: 10.1137/1.9781611972931.10.

[26] E. F. D. O. Sandes, A. Boukerche, and A. C. M. A. De Melo, "Parallel Optimal Pairwise Biological Sequence Comparison," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 1–36, Mar. 2016, doi: 10.1145/2893488.

[27] J. Hoffmann, D. Zeckzer, and M. Bogdan, "Using FPGAs to Accelerate Myers Bit-Vector Algorithm," 2016, pp. 535–541, doi: 10.1007/978-3-319-32703-7_104.

[28] H. Hyyrö and G. Navarro, "Faster Bit-Parallel Approximate String Matching," 2002, pp. 203–224, doi: 10.1007/3-540-45452-7_18.

[29] D. N. S. S. Liyanage, G. V. M. P. A. Fernando, D. D. M. M. Arachchi, R. D. D. T. Karunathilaka, and A. S. Perera, "Utilizing Intel Advanced Vector Extensions for Monte Carlo Simulation based Value at Risk Computation," *Procedia Comput. Sci.*, vol. 108, pp. 626–634, 2017, doi: 10.1016/j.procs.2017.05.156.

[30] "Intel® 64 and IA-32 Architectures Software Developer Manuals," *Intel Software Developer Zone*, 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm.

[31] P. Gepner, "Using AVX2 Instruction Set to Increase Performance of High Performance Computing Code," *Comput. Informatics*, vol. 36, no. 5, pp. 1001–1018, 2017, doi: 10.4149/cai_2017_5_1001.

[32] D. Kusswurm, *Modern X86 Assembly Language Programming*, 2014, doi: 10.1007/978-1-4842-0064-3.

[33] "Intel® 64 and IA-32 Architectures Optimization Reference Manual," *Intel Corporation*, 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf. [Accessed: 12-Oct-2019].