DLSU Research Congress 2021
De La Salle University, Manila, Philippines
July 7 to 9, 2021

Multiprocessing Implementation of Pigeonhole-based Filter for DNA Read Mapping

Roger Luis Uy[1*], Aaron Russell Fajardo[2], Candace Claire Mercado[2], Raphael Zapanta[2], and Saira Kaye Manalili[2]
[1] *Computer Technology Department, College of Computer Studies, De La Salle University*
[2] *College of Computer Studies, De La Salle University*
**Corresponding Author: roger.uy@dlsu.edu.ph*

Abstract: Filter-verification technique is one of the methods used in mapping DNA reads to some known reference genomes. The key concept is to filter out dissimilar sequences while the rest are forwarded to the computationally intensive verification step. Specifically, pigeonhole-based filter partitions a read into several seeds based on the given value of q (i.e., q-gram) and searches the locations of each seed on the reference genome. To minimize repetitive traversing of the large reference genome during seed matching, a q-gram index hash table is generated. This table contains all occurrences of the q-grams present in the reference genome. There are three methods for building an index-based hash table: direct addressing, open addressing and minimizer-based addressing. In this study, the pigeonhole-based filter is implemented with two modes, namely (1) match all seeds and (2) match seeds and exit immediately once the read acceptance threshold has been met. To accelerate the filter process and to exploit the multi-core architecture, the pigeonhole-based filter is implemented using multiprocessing. The objective of this study is two-fold: to perform a comparison analysis between the sequential and multiprocessing implementation and to perform comparison of the pigeonhole filter vis-à-vis the three index-based hash table. Based on the results, the parallel implementation is faster than its sequential counterpart by a factor ranging from 2.12 to 3.24. Result also shows that open addressing paired best with pigeonhole-based filter using exit mode with q value greater than 14.

Key Words: Bioinformatics; Read mapping; Pigeonhole Principle; Filter-verification

1. INTRODUCTION

The Next Generation Sequencing (NGS) technology has become a breakthrough in the field of bioinformatics. This provided researchers (Egan, Schlueter, & Spooner, 2012; Deurenberget al., 2017) a new method to understand biological and biomedical questions. This platform generates large number of short DNA fragments called reads. These reads are then mapped to the reference genome to determine the location and its relation to the reference genome. This process is called read mapping. Early read mapping process uses dynamic programming algorithms to determine similarity score between the sequences (Levenshtein, 1966; Needleman & Wunsch, 1970; Smith & Waterman, 1981). But due to the quadratic time complexity of the algorithm and the large amount of data to be processed makes those algorithms computationally costly.

To improve the read mapping process, later works (Weese, Holtgrewe, & Reinert, 2012; Alkan et

al., 2009; David, Dzamba, Lister, Ilie, & Brudno, 2011) use the filter-verification paradigm. In the filter stage, a read is partitioned into seeds (i.e., q-grams). Each seed is then compared to the reference genome. A read is accepted or rejected based on the number of accepted seeds. The criteria for the number of accepted seeds are based on the generalized pigeonhole principle.

The generalized pigeonhole principle states a read is accepted if at least one seed from the read has at most $\lfloor e/j \rfloor$ edits, where e, the error threshold, is a user defined integer value, and j is the number of partitions. Those reads in which seeds are not found on the reference genome will be filtered out, while the rest will proceed for verification.

To speed up the process of filtering, a q-gram index hash table of the reference genome is built. The hash table contains the location of each q-gram with respect to the reference genome. This is to reduce the repetitive scanning the location of each seed to the reference genome (Canzar & Salzberg, 2017). There are three methods for building an index-based hash table: direct addressing, open addressing and minimizer-based addressing.

In order to make the filtering process faster, the parallelization of the process can be implemented by utilizing the multi-core system. A method of taking advantage of the multi-core system is called multiprocessing. Multiprocessing involves two or more CPU cores executing different tasks. The parallel implementation of the pigeonhole filter can be done by distributing the seed selection in each CPU core.

Numerous works have implemented the hash-based method of pre-alignment filtering include RazerS3 (Weese, Holtgrewe, & Reinert, 2012), mrFast (Alkan et al., 2009), and SHRiMP2 (David, Dzamba, Lister, Ilie, & Brudno, 2011). From the works mentioned, some were able to implement in multiprocessing but none of the works were able to compare which index-based hash table build method is best suitable for pigeonhole filter. The contribution of this study is two-fold: to perform a comparison analysis between the sequential and multiprocessing implementation and to perform comparison of the pigeonhole filter vis-à-vis the three index-based hash table.

The outline of the rest of the paper is as follows: section 2 discusses the implementation of pigeonhole-based filter. Section 3 discusses the reference genome and read datasets used. Section 4 discusses the comparison of run time between sequential and parallel implementation of pigeonhole filter using varying values of q. Lastly, observation of the results is discussed in Section 5.

2. PIGEONHOLE-BASED FILTER DESIGN

The pigeonhole-based filter process starts by partitioning the read into seeds depending on the value of q used by the index hash table. There are instances where the length of the seed is not equal to the value of q. To solve this problem, adjustment of the position is made by adding more base pairs from the left of the starting position of the seed. Then, each seed will be searched vis-a-vis with the reference genome. The seed will be accepted if a location if found on the reference genome.

After all the seeds are checked, the read acceptance value is checked. The criteria for the read acceptance value are the following:

- If $e < q$, then
  $$acceptanceValue = (j - e)$$
- If $e \geq q$, there are two conditions:
  - If $\lfloor e/j \rfloor < q$, then
    $$acceptanceValue = j - \left\lfloor \frac{e}{(allowableError + 1)} \right\rfloor$$
  - If $\lfloor e/j \rfloor \geq q$, then
    $$acceptanceValue = j - \left\lfloor \frac{e}{allowableError} \right\rfloor$$

The $allowableError$ is computed by $\lfloor e/j \rfloor$, which is based on the generalized pigeonhole principle. Please refer to figure 1 illustration.

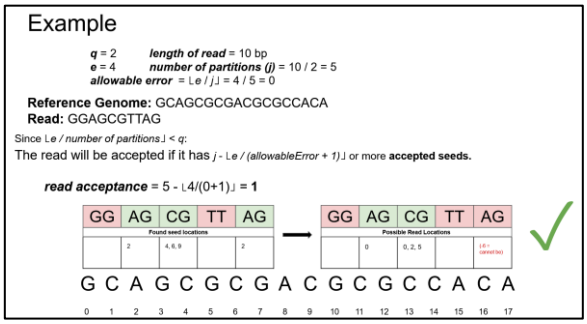

Fig. 1. Example of pigeonhole-based filter for use case where $e \geq q$ and $\lfloor e/j \rfloor < q$

The filter has two modes: search all seeds and check if the number of accepted seeds have met the read acceptance value, and exit searching immediately if the number of accepted seeds have met the read acceptance value.

The indexing method used in this study is hash-based index. The hash-based indexing method

uses rank or hashed value of the seed, and the hashed value is used to determine the location of the seed on the reference genome. Three methods of building hash-based index are direct addressing, open addressing, and minimizer-based q-gram indexing method.

Direct addressing q-gram index method uses two tables: directory table, and position table. The directory table contains the starting position of each q-gram in the position table. The position table contains the starting location of the q-gram on the reference genome.

Since not all q-grams need to have an entry in the directory table, the open addressing method adds a new table called code table where each q-gram is hashed into. This maps only the q-gram present in the reference genome to their respective positions in the directory table.

To minimize the storage requirements of q-gram index hash table, the minimizer-based q-gram index stores the representative q-gram, known as minimizer, from each window of w consecutive q-grams. Consecutive means that each q-gram contained by that particular window is shifted to the right by one character.

The pigeonhole filter was implemented in C++. An application programming interface (API) for multiprocessing called OpenMP was used to utilize the multi-core processor. Each thread of the CPU core handles different sets of read to be processed. The testing platform used is Intel i7-6700HQ (4-core) 2.60 GHz with 12GB RAM and running on Ubuntu 19.04 operating system.

3. METHODOLOGY

In the parallelization of the seed selector, the code will utilize all the CPU cores and each core will have a seed selector (see figure 2). In this way, it will check four reads at the same time assuming the CPU has four cores. The code for implementing the parallelization is provided by OpenMP by using the provided preprocessor directives. The directive, *#pragma omp parallel for* states the whole for-loop process is parallelized. It is also important to state the critical section by placing *#pragma omp critical* to avoid race conditions.

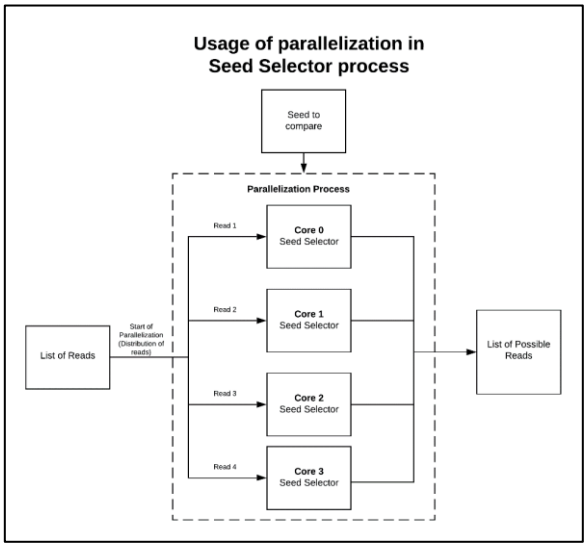

Fig. 2. Parallel implementation of seed selector

There are two reference genomes used in this study: synthetic data containing 8 million base pair (bp), and *E. coli* real dataset with 5.4 million bp. The synthetic data contains random generated DNA nucleotides. As for the read datasets, these were generated from reference genomes. The read datasets consist of 1,000,000 reads with a length of 150 bp in each read.

To determine the performance of the pigeonhole filter, both sequential and parallel implementation run times were recorded. Except for the direct addressing method, the q values used are 10, 12, 14, 16, 18 and 20. Given memory limitations, direct addressing can only execute up to a q-gram size of 14. The filter was tested using its two modes (i.e., search all mode, and exit found mode) and with varying values of error threshold to determine if this affects the run time and the number of locations found.

4. RESULTS

Varying values of e were used to determine the effect in the seed selector. Section 4.1 discusses the result of using $e = 0$ or accepting reads with no errors.

Section 4.2 discusses the result of using $e > 0$ or accepting reads with errors. The values of e are dependent to the value of q. The computed values of e were based on the middle, and maximum values of the seeds of the read.

4.1 Using e = 0

Referring to figures 3 and 4, there is no significant difference in the run time when using search all mode and exit found mode. This is because the computed read acceptance value is the same in both modes; thus, regardless of the mode, all the seeds from each read were checked.

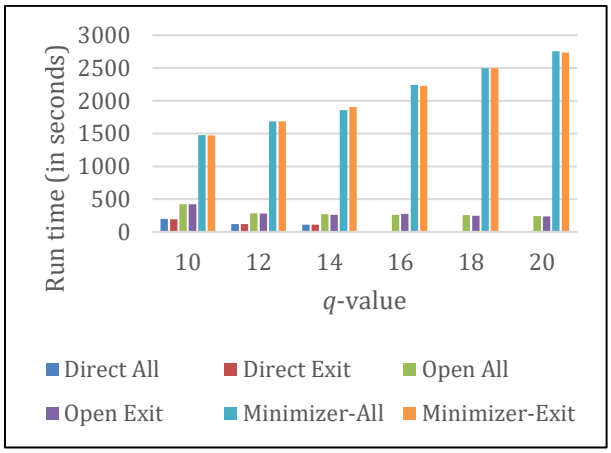

Fig. 3. Pigeonhole filter sequential runtime on synthetic dataset across various *q* for the 3 hash tables

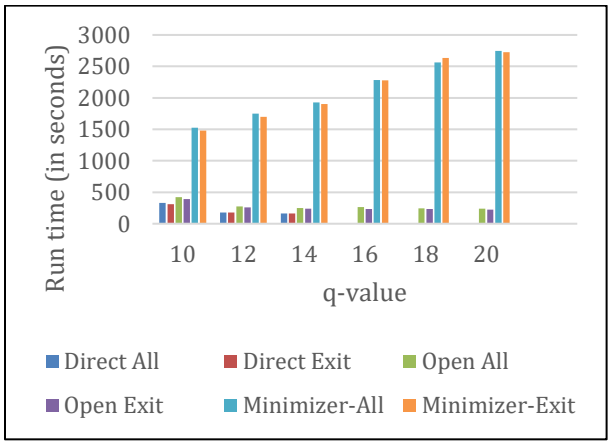

Fig. 4. Pigeonhole filter sequential runtime on *e.coli* dataset across various *q* for the 3 hash tables

The run time for direct addressing and open addressing is inversely proportional to the value of *q*. In the case of minimizer-based, the run time is directly proportional to the value of *q*. Even though using direct addressing is faster but due its high memory consumption, in can only handle up to *q*=14. As such, open addressing method is best paired with pigeonhole filter with *q* value greater than 14.

Referring to figures 5 and 6 and comparing it with figures 3 and 4, the speed up of synthetic dataset using parallel over sequential implementation factor is 2.27 to 3.03 for direct; 2.75 to 3.03 for open and 2.15 to 3.24 for minimizer-based method. Similarly, the speedup using *E. coli* dataset ranges from 2.29 to 2.87 across three methods. Thus, the parallel implementation of the seed selector is 2 to 3 times faster than the sequential implementation.

By using open addressing, coupled with *q* value greater than 14 is the best combination for pigeonhole filter.

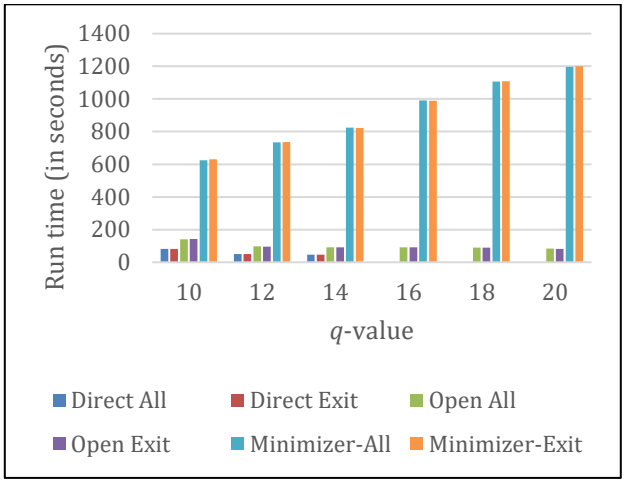

Fig. 5. Pigeonhole filter parallel runtime on synthetic dataset across various *q* for the 3 hash tables

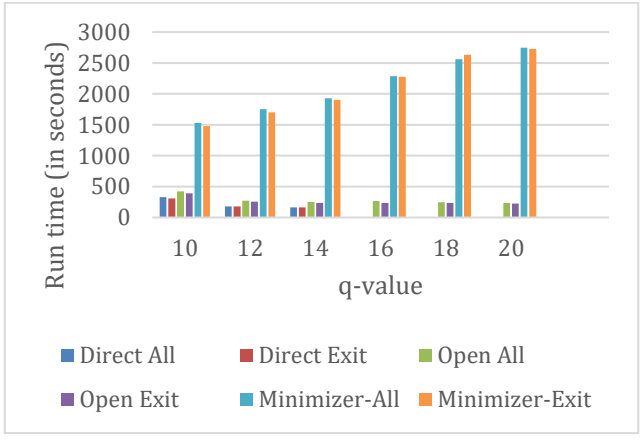

Fig. 6. Pigeonhole filter parallel runtime on *e.coli* dataset across various *q* for the 3 hash tables

4.2 Using e > 0

Referring to figures 7 and 8, As opposed to e=0, there is a difference in the run time of search all mode as compared to exit mode for e>0. The reason is that exit mode stops checking the rest of the seeds if it already met the read acceptance value. This results in faster run time.

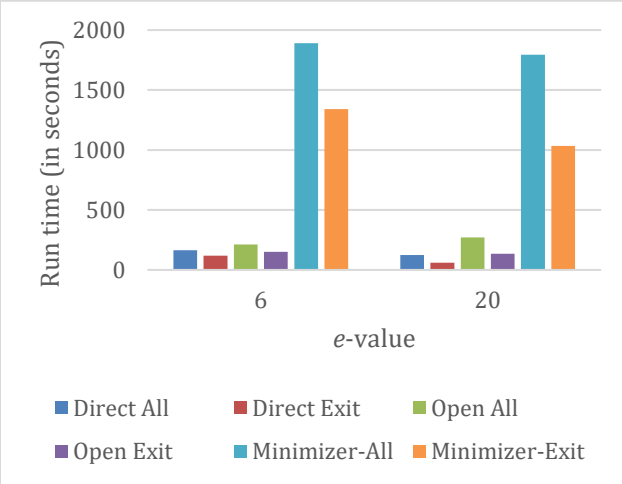

Fig. 7. Pigeonhole filter sequential runtime on synthetic dataset across various e for the 3 hash tables for q=14

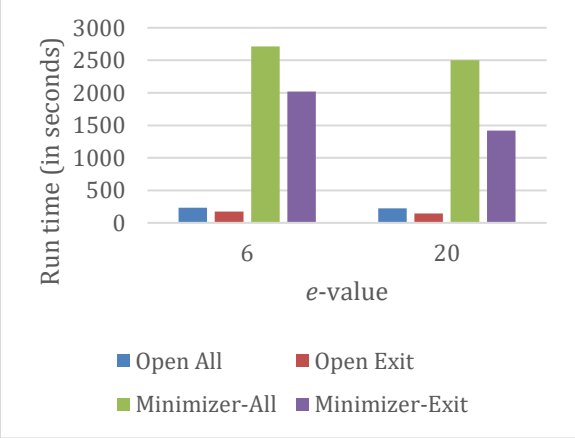

Fig. 8. Pigeonhole filter sequential runtime on sythetic dataset across various e for the 3 hash tables for q=20

As with e=0, minimizer-based hash table is consistently slower than the rest of the hash tables. As mentioned previously, there is no runtime value for direct hash table value for q>14.

5. CONCLUSION

Using direct addressing is the fastest among the hash-based indexing methods. However, due to memory limitation, it can only handle up to q = 14. For practicality, the direct addressing method is not recommended. The run time difference between direct and open addressing is minimal. Using minimizer-based is the slowest among the other hash-based indexing methods. The run time for minimizer-based is directly proportional to the value of q unlike direct and open addressing which are both indirectly proportional to the value of q. In terms of the search modes, exit mode is faster than search all mode if the error threshold is more than 0.

.

Overall, it is recommended to pair pigeonhole-based filter in exit mode using open addressing hash-based index with q value greater than 14.

6. ACKNOWLEDGMENTS

We would like to thank the Bioinformatics Laboratory, headed by Dr. Anish MS Shrestha, for allowing us to utilize the laboratory's computing environment.

7. REFERENCES

Alkan, C., Kidd, J.M., Marques-Bonet, T., Aksay, G., Antonacci, F., Hormozdiari, F., Mutlu, O. (2009). Personalized copy number and segmental duplication maps using next-generation sequencing. Nature Genetics, 41(10), 1061.

Canzar, S., & Salzberg, S. L. (2017, March). Short Read Mapping: An Algorithmic Tour. Proceedings of the IEEE, 105(3), 436–458.

David, M., Dzamba, M., Lister, D., Ilie, L., & Brudno, M. (2011). SHRiMP2: sensitive yet practical SHort Read Mapping. Bioinformatics, 27(7), 1011–1012.

Deurenberg, R. H., Bathoorn, E., Chlebowicz, M. A., Couto, N., Ferdous, M., Garcia-Cobos, S., Rossen, J. W. (2017). Application of next generation sequencing in clinical microbiology and infection prevention. Journal of Biotechnology, 243, 16–24.

Egan, A., Schlueter, J., & Spooner, D. (2012). Applications of next-generation sequencing in plant biology. American journal of botany, 99,175-85.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions and Reversals. Soviet Physics Doklady, 10, 707.

Needleman, S. B., & Wunsch, C. D. (1970, March). A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology, 48(3), 443–453.

Smith, T.F., & Waterman, M.S. (1981). Identification of common molecular subsequences. Journal of Molecular Biology, 147(1), 195–197.

Weese, D., Holtgrewe, M., & Reinert, K. (2012). RazerS3: Faster, fully sensitive read mapping. Bioinformatics, 28(20), 2592–2599.

Xin, H., Lee, D., Hormozdiari, F., Yedkar, S., Mutlu, O., & Alkan, C. (2013). Accelerating read mapping with FastHASH. BMC Genomics, 14(S1), S13.