# On Performance Evaluation of BM-Based String Matching Algorithms in Distributed Computing Environment

Kunaphas Kongkitimanon and Boonsit Yimwadsana

***Abstract*—String matching algorithms plays an important role in many applications of computer science: in particular searching, retrieving and processing of data. Various fields that rely on computer science for computing and data processing such as science, informatics (e.g. biology, medical, and healthcare), statistics, image, video/signal processing and computational aspect of business (e.g. finance, accounting, and computer security) would benefit greatly from efficient data search algorithm, in particular string matching. Any applications involving the use of database would use string matching algorithm. Many string matching algorithms such as TBM (Turbo Boyer Moore), BMH (Boyer-Moore-Horspool), BMHS (Boyer Moore Horspool Sundays, and BMHS2 (Boyer Moore Horspool Sundays 2) were introduced based on the celebrated BM (Boyer-Moore) algorithm considered to be one of the early efficient string searching algorithms. Although these algorithm offers significant performance improvement over the BM algorithm, they were designed with the assumption of single core computer architecture which executes the algorithm in a serialized manner. Today, multiple-core-processor computers are very common, and applications are designed to process big data thanks to the advanced in computing technology of various fields. High performance computing system utilizing parallel and distributed computing has started to become popular. This work evaluates and compares the performance of the aforementioned string matching algorithms in parallel and distributed environment for high performance computing with respect to that of the serialized single-core computing platform. In this work, the variants of BM algorithms are implemented and evaluated on Apache Spark, a popular distributed computing platform, by executing a set of queries of different search pattern lengths.**

***Index Terms*—Apache spark, Boyer Moore, distributed computing, string matching.**

## I. Introduction

String matching algorithms have been used greatly in computing for various kinds of applications. One of the most important applications is database query which is used universally throughout the world for data processing and management. Thanks to current technologies that have the ability to collect more data than ever before, string matching in big data could be a major bottleneck in many applications. If not done efficiently, a typical database query using "LIKE" command can take up to 10 minutes to search for a DNA pattern in a regular human genome sequence data without indexing. The "MATCH" command, recently added to MySQL database server [1] relies on the hashing of the entire words, but it is unable to search for patterns inside word strings.

Thanks to the development of parallel and distributed computing platform such as Hadoop and Apache Spark, search queries could be performed quickly depending on the number of nodes and resources of the nodes in the system. However, parallel and distributed computing is still considered a new concept and string matching algorithms were not originally designed to directly support parallel and distributed computing platform. Hence, we are interested in finding out whether the performance of string matching algorithms based on Boyer-Moore (BM) algorithm can perform as same as we expect for single-core computing environment.

This work contains 8 parts including introduction, background, hypothesis, experiment setup, results and discussion, conclusion, future work and acknowledgement.

Manuscript received January 9, 2017; revised March 19, 2017. This research project was supported by the Faculty of Information and Communication Technology, Mahidol University and the Integrative Computational Bioscience Center, Mahidol University.

The authors are with Faculty of Information and Communication Technology and Integrative Computational Bioscience Center, Mahidol University, Thailand (e-mail: kunaphas.kon@gmail.com, boonsit.yim@mahidol.ac.th).

## II. Background

In this paper, we consider five algorithms which are efficient variants of BM algorithm. We briefly explain each algorithm as follows:

### A. Boyer-Moore (BM) Algorithm [2]

In contrary to the fundamental programming technique that searches for matched string by comparing character at a time from leftmost to rightmost position (brute-force) and slide the comparison window forward one character at a time, the BM algorithm compares characters in the comparison window in reverse order (from rightmost to leftmost) of the pattern. In addition, if a character mismatch is found during the comparison, the comparison window will be shifted. In order to determine shifting value when a mismatch or a complete match occurs, it requires two pre-processing rules which are "good suffix" and "bad character". In pre-processing phase for calculating bad character shifts, the time complexity is O (m+σ), where σ is the size of the finite character set relevant with pattern and text. The best case of Boyer-Moore string matching time is O(n/m) compared to the worst case which is O(mn) where m is the size of pattern and n is the size of text to be searched.

*Advantage:* The combination of both good suffix and bad character rule provides a good shift value.

*Disadvantage:* The preprocessing of Good suffix rule is complex to Implement and understand. Bad character rule

doi: 10.18178/ijfcc.2017.6.1.479

may produce small shift

### B. Turbo Boyer Moore (TBM) [3]

TBM algorithm is a variation of the Boyer-Moore algorithm. It remembers and recognizes the substrings of the text that have already been matched against the pattern since previous comparisons so that it will not compare the matched substrings again for the next iteration. Then, TBM will just compare the leftover characters of the pattern with the text to be matched. TBM algorithm uses the same good suffix and bad character rules as BM algorithm, but it requires an extra space in order to remember the text that matched a suffix of the pattern during the last attempt. The preprocessing phase for calculating bad-character shifts can be performed in O(m+n) time and space complexity. The searching phase is in O(n) time complexity.

*Advantage*: It can sometimes perform a turbo-shift which skips matched substrings found from previous comparison.

*Disadvantage:* Sometimes, TBM may produce small pattern shift value because the turbo-shift is used only when a good-suffix shift was performed [4].

### C. Boyer Moore Horspool (BMH) [5]

BMH algorithm does not use the same shifting methods as BM algorithm. It removes the good suffix rule because the good suffix rule is not compatible with its search window shifting approach. The BMH algorithm uses only the bad character rule in order to maximize the length of the shifts by ignoring the location of character-mismatch locations prior to the last character of the pattern. The shift distance is determined by first determining whether the rightmost character in the text string region in the comparison window appear in the pattern search string. If the last character is not found in the pattern, the position of the pattern for the next search iteration will be the length of the pattern. Otherwise, the shift value will be the relative position of the matched character in the pattern. Preprocessing time complexity is O(m+n) and searching time complexity is O(mn)

*Advantage*: The concept of good suffix rule is removed so that the algorithm can be implemented to support its basic concept. In case of mismatch, the shift value is determined by the bad character value of last character instead of character that caused mismatch so more jump is archived.

*Disadvantage*: The removal of good-suffix sometimes may not give the shift as much as that in BM

### D. Boyer Moore Horspool Sunday (BMHS) [6], [7]

The basic idea of BMHS algorithm is to extend the BMH algorithm by considering the next-to-last (next-to-rightmost) character of the current text in the comparison window (T[m+1], where m is the relative position to the first character of the pattern in each comparison iteration) instead of considering the last character of the current text in the comparison window (T[m]). If the next-to-last character, T[m+1], does not appear at all in the pattern, the shift value for the comparison window is pattern length plus 1. Otherwise, it calculates the shifts just like BM algorithm by using the Bad suffix rule. This algorithm increases the shift value by 1 compare to BMH algorithm.

*Advantage:* BMHS offers the current largest value of shift value which is equal to pattern length plus one.

*Disadvantage*: Suppose the last character of the text in the current comparison window aligning to the last character of pattern is a mismatch, but next-to last character appears in the pattern string, the shift value is larger in BMH algorithm than in BMHS algorithm.

### E. Boyer Moore Horspool Sunday 2 (BMHS2) [7], [8]

The idea of this algorithm is similar to BMHS that when a single mismatch occurs at a position then the algorithm considers the next-to-last character and last character of the corresponding text, T[i+m+1] and T[i+m] respectively, where m is the length of the pattern. The shift value for the next pattern search position is calculated based on following rules

1) If the current next-to-last character is not in the pattern, right shift the pattern by m+1.
2) If the current next-to-last character occurs at the first position of the pattern, right shift the pattern by m.
3) If the next-to-last character occurs at any position other than the first position of pattern, then it will look at the previous next-to-last character in the text to find out whether this character appears in the pattern or not. If the character is found in the pattern, the shift will be determined according to the bad character rule. Otherwise, the shift value is m+1, because it no longer needs to compare anything else since the character of previous next-to-last character does not appear together with the next-to-last character. As a result, the whole pattern will not be matched.

*Advantage*: This algorithm considers last character and next-to-last character both so it combined advantages of both BMH and BMHS.

*Disadvantage:* The overhead from character search increases as we have to search two characters instead of just one character in the case of BMH and BMHS in order to determine the shift value.

### F. Apache Spark [9], [10]

Apache Spark is a "big data" processing framework. It was developed in 2009 at UC Berkeley's AMP Lab. Apache Spark is generally a lot faster than Hadoop MapReduce [10] because of using in-memory data storage and the way its processes the data . It processes "big data" on distributed data collections calls RDD (resilient distributed dataset) which are stored in distributed memory cluster. Spark holds intermediate results in physical memory in contrast to Hadoop which needs to write immediate results to disks and then read updated data from the disks in order to perform the next operation [11]. In some cases, Spark enables application clusters to run up to 100 times faster than Hadoop MapReduce, and 10 times faster even when running on disk [12]. Spark also support many big data analytics and machine learning applications such as GraphX, MLlib, and SQL. However, Spark does not come with its own file management system, so it needs to be integrated with others file management system, for example HDFS (Hadoop Distributed File System). Spark currently support three types of cluster managers: standalone, Apache Mesos and Hadoop YARN.

### G. Running on a Cluster [13]-[15]

In distributed mode, Spark use a master/slave architecture as shown in Fig. 1 with one central coordinator, Spark Driver, and many distributed workers which call Spark Executors. The driver and each of the executors run in their own Java processes. Spark applications run processes on cluster independently, and it worked in coordination with SparkContext in the main program which call a driver program as an application interpreter.

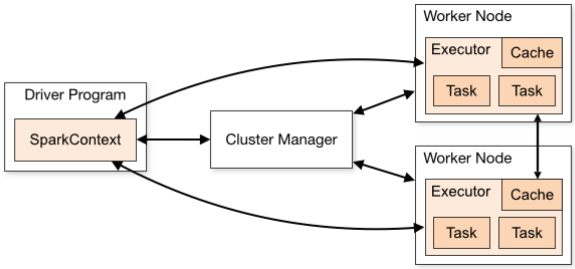


Fig. 1. The components of a distributed Spark application [13].

To run on cluster, an application is submitted to the cluster via spark-submit command. Spark-submit launches the driver program and invokes the main function specified by the user. The driver program instantiates a SparkContext instance. The SparkContext connects to the cluster manager in order to allocate resources across cluster. The cluster manager starts executors, which are processes that run computations and store data for the application. Next, the driver performs two duties; converting a user program into tasks and scheduling tasks on executors. After finished, it sends application code (work) to the executors in form of tasks. Executors compute the tasks and save the results.

## III. Hypothesis

We believe that in distributed computing environment (Apache Spark), the variants of BM-based algorithm may not produce the same comparable performance that we observe in single-core computing environment.

## IV. Experiment Setup

### A. Experimental Environment and Datasets

All experiments were carried out in a cluster of CentOS virtual machines consisting the maximum four slave nodes, one driver program node and one master node. Each node has 2 processing cores (Intel core i7-6700 @ 3.40 GHz) and 6 GB of physical memory assigned, thus the total number of cores in the system is 12. The operating system for the cluster is CentOS 7 running Apache Spark standalone cluster 2.0.0.

The datasets include the unbiased 1-GB dump text content of Wikipedia [16] in XML format, and 3.2-GB HG19 Human genome reference file [17].

### B. Implementation

The BM-variant algorithms were implemented in Java. Each algorithm accepted two parameters: text to be searched and pattern. The codes were called by Apache Spark through Scala programming interface. We defined the base Resilient Distributed Datasets (RDD) from the external datasets which we used as input. Next, we created the data map transformation (as a part of Map-Reduce operation) by passing the BM-variant algorithm codes in data map and return the results in pair of line and pattern found. Then we ran our Spark application under different scenarios as follows:

For the 1-GB text data extracted from Wikipedia, we used 3 text patterns in different lengths which were “cat”, “Thailand”, and “recommendations”. For the 1-GB human genome data, we used 3 text patterns in different lengths which were “TAT”, “TTTGCGGTAAG”, and “AGAACGCAGAGACAAGGTTC”.

## V. Result and Disscusion

### A. Experiment Results

Firstly, we tested the variants of BM algorithm by using the following pattern sets: “cat, “Thailand” and “recommendations” search against the 1-GB Wikipedia file. We ran each algorithm for 5 times and collected the average of the computing time under a single machine computing environment as shown in Fig. 2.

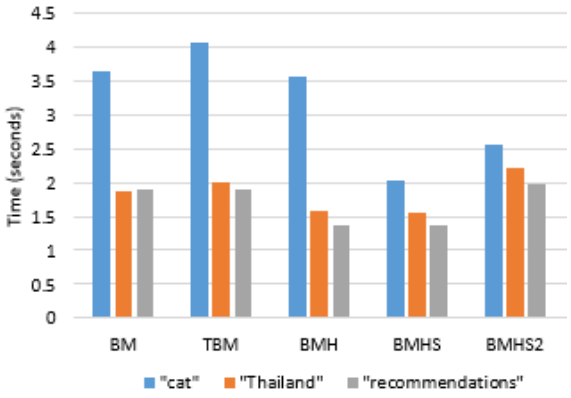


Fig. 2. The computing time of variants of BM algorithm using common English word patterns: “cat”, “Thailand”, and “recommendations” against 1-GB Wikipedia data.

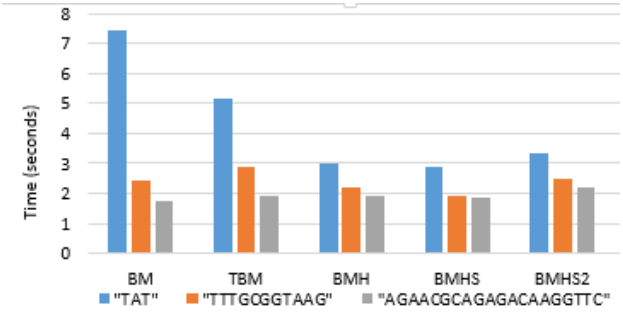


Fig. 3. Computing times of the variants of BM algorithm using random DNA sequence string “TAT”, “TTTGCGGTAAG” and “AGAACGCAGAGACAAGGTTC” against 1-GB Human Genome data

According to Fig. 2, the performance of BMHS was the best overall. This is due to its ability to perform the maximum size of comparison-window shifts (the length of the pattern plus one or m+1) on average as explained in Section II. BMHS2 did not perform well even though it was claimed to be an improvement over BMHS. We found that this is due to its lack of preprocessing for bad character. It has to compute

the same bad character rule method every single time in order to determine a pattern shift value. The performance of BMH and BMHS are similar when the length of the pattern is large.

According to Fig. 3, all BM-variant algorithms performed quite well for DNA data search using long-length patterns in single machine environment. This is because all BM-variant algorithms can take advantage of higher character matching probability for bad character rule when the character space is small (e.g., 'A', 'T', 'G', and 'C').

In distributed computing environment (Apache Spark), according to Fig 4 the performance of BMHS was the worst, and BMHS2 was the best among all BM-based variants. The performance shown in Fig 4 was contrary to what we expected.

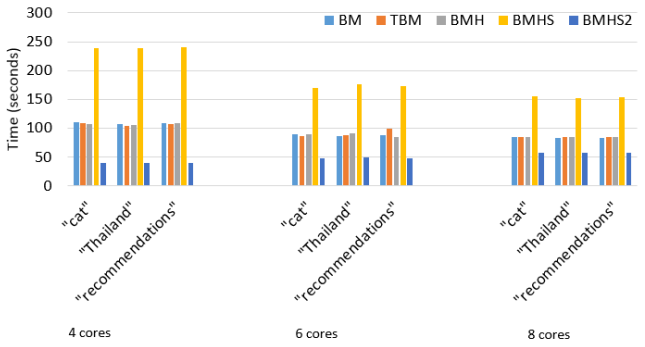


Fig. 4. The computing times of variants of BM algorithms using common English word patterns: "cat", "Thailand", and "recommendations" against 1-GB Wikipedia data in Apache Spark environment.

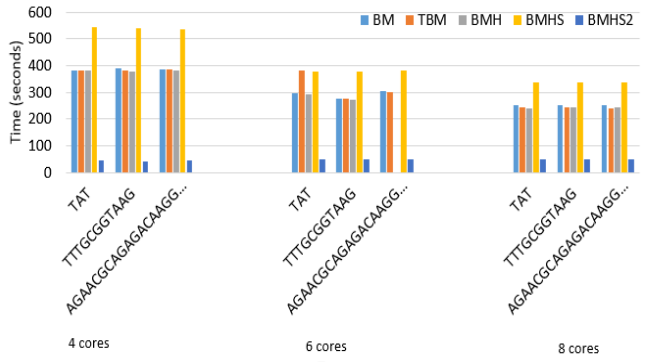


Fig. 5. The computing times of variants of BM algorithms using random DNA sequence string "TAT", "TTTGCGGTAAG" and "AGAACGCAGAGACAAGGTTC" against 3.2-GB Human Genome data in Apache Spark environment.

In Fig. 5, BMHS2 performed the best among variants of BM algorithms. This is because its implementation does not construct an array for bad character rule in preprocessing stage in every Spark task. The construction of this array requires large amount of extra space because of Spark behavior on task distribution. An Apache Spark driver program serializes a function and submits it to executors, and then the executors will execute the code along with their own partitions. The other BM-based variants except BMHS2 have a preprocessing stage which calculates a shift value and store in a bad character array. As a result, the preprocessing overhead has to be performed for each task rather than one single time at the start of the whole operation.

We investigated the computing environment and datasets further in order to determine whether this surprising performance is due to the characteristics of datasets (which should not be any), how Apache Spark executed our codes for BMHS, or how Apache Spark processed the datasets. We found that the performance issue may be related to how Apache Spark processed our datasets because Apache Spark read the datasets one line at a time [18]. In addition, Apache Spark waits until each node finishes computation for each block of lines in order to aggregate the results before continuing to process the next block. Hence, a large number of 'newline' characters, '\n', that a dataset contains, affects the performance of Apache Spark. If each line contains short text, BMHS will not be able to fully take advantages of their large pattern shift value because the number of times that a maximum shift (m+1) can occur is low.

We confirmed our explanation by running the experiments excluding the 'newline' characters from the dataset in different portions: 25, 50 and 75 percent. The performance of the variants of BM-based algorithms is shown in Fig 6

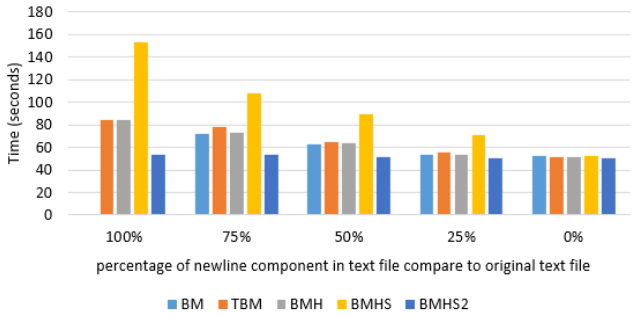


Fig. 6. Computing times of the variants of BM algorithm on 8 cores by using "Thailand" as a pattern search against different portions of 'newline' in Apache Spark environment.

After we understood how 'newline' characters affect the performance of Apache Spark and BM-based algorithms, we completely clean our datasets and run the same experiment again in different resource settings of Apache Spark. Fig 7 shows the performance of the variants of BM-based algorithms on dataset with no 'newline' character in systems with different number of processor cores.

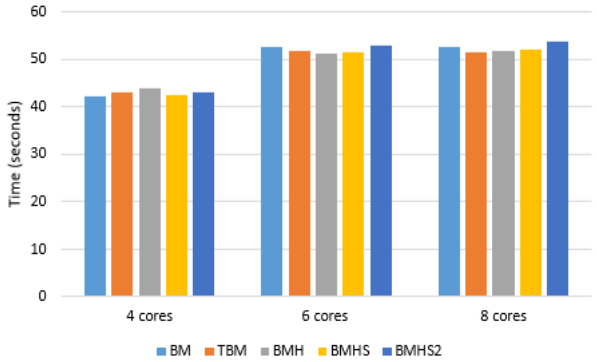


Fig. 7. The computing times of variants of BM algorithms using common English word patterns "Thailand" against 1-GB cleaned data (zero 'newline' character) of Wikipedia data in Apache Spark environment.

We observed that there is no performance difference among all variants of BM-based algorithms in distributed computing environment using Apache Spark when we increase numbers of cores and use the dataset with no 'newline' character. This is because when the text is transformed to contain only one single line, there is no distributed computing process occurs. Apache Spark

distributes tasks one line at a time. If the data only contains one single line, only one single node will process this entire dataset. We confirmed this by monitoring the task and executor characteristics using Apache Spark Web UI. The results are shown in Table I.

TABLE I: THE TABLE SHOWS THAT ONLY ONE NODE EXECUTES THE WHOLE DATA SET. (THIS RESULT GATHERED FROM TASK TABLE ON SPARK WEB UI)

| Executor ID | Input size / Records |
|---|---|
| 1 / 10.22.52.208 | 0.0 B / 0 |
| 0 / 10.22.52.206 | 0.0 B / 0 |
| 2 / 10.22.52.207 | 0.0 B / 0 |
| 3 / 10 .22.52.210 | 965.0 MB / 0 |

Moreover, when we submitted an application in Apache Spark cluster mode, the Apache Spark driver had to coordinate with workers and overall execution of tasks frequently, so the performance of each algorithm is limited by these actions in order to execute the task which is same as how single machine execute a task (search the whole text file without waiting command from driver like Apache Spark in order to proceed next execution).

## VI. CONCLUSION

This paper discussed and showed the performance of the variants of BM-based algorithms on a single machine, and the Apache Spark distributed computing environment in cluster mode. The performance of the variants was we expected according to the literature review described in Section II. When we performed the experiments in Apache Spark distributed computing environment, we found that BMHS2 performed the best among other variants of BM algorithms. Additionally, the number of 'newline' characters greatly affect the performance of BMHS compare to other variants due to the way that Apache Spark read the data and process the code. After removing the 'newline' characters, the performance of all variants of BM-based algorithms was not significantly better than single machine and varied based on the amount of resources given to the cluster nodes. However, there is no performance difference among all variants of the BM-based algorithms. It is important to note that once the datasets are cleared of newline characters, all BM-based variants perform almost the same in Apache Spark distributed computing platform because only one single node will be selected by the task schedule to determine when to run the task. This is not true in single machine computing platform whereas BMHS performs the best.

## VII. FUTURE WORK

We will focus on improve variant BM algorithms to create an algorithm that can run well in single-machine environment and more efficiently on distributed computing environment (e.g., Apache spark).

## ACKNOWLEDGMENT

This research project was supported by Faculty of Information and Communication Technology, Mahidol University and the Integrative Computational Bioscience Center, Mahidol University, Bangkok, Thailand.

## REFERENCES

[1] Comparison of b-tree and hash indexes. [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html
[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM* , vol. 20, no. 10, pp. 762-772, 1977.
[3] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Speeding up two string matching algorithms," *Algorithmica*, vol. 12, pp. 247-267, 1994.
[4] C. Charras and T. Lecroq. Turbo-BM algorithm. [Online]. Available http://www-igm.univ-mlv.fr/~lecroq/string/node15.html
[5] R. N. Horspool, "Practical fast searching in strings," *Software-Practice and Experience*, vol. 10, pp. 501-506, 1980.
[6] D. M. Sunday, "A very fast substring search algorithm," *Communications of the ACM*, vol. 33, no. 8, pp. 132-142, 1990.
[7] R. Choudhary, A. Rasool, and N. Khare, "Variation of Boyer-Moore string matching," *International Journal of Computer Science and Information Security*, vol. 10, no. 2, pp. 95-101, 2012.
[8] L. Xie, G. Yue, and X. Liu, "Improved pattern matching algorithm of BMHS," in *Proc. the 2010 Third International Symposium on Information Science and Engineering*, Washington DC, pp. 616-619, Dec 2010.
[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. the 2nd USENIX Conference on Hot Topics in Cloud Computing*, Boston, pp. 10-10, June 2010.
[10] S. Penchikala. (2015). Big data processing with apache spark. [Online]. Available: https://www.infoq.com/articles/apache-spark-introduction
[11] K. Noyes, "Five things you need to know about Hadoop vs. Apache Spark," IDG News Service, December 2015.
[12] Apache spark-lightning-fast cluster computing. [Online]. Available: http://spark.apache.org/
[13] cluster-overview. [Online]. Available: http://spark.apache.org/docs/latest/cluster-overview.html
[14] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark Lightning-Fast Data Analysis*, 1st Ed. O'Reilly Media, 2015 ch. 7, pp. 117-140.
[15] What are workers executors cores in spark standalone cluster. [Online]. Available: http://stackoverflow.com/questions/32621990/what-are-workers-executors-cores-in-spark-standalone-cluster
[16] M. Mahoney. (2011). Mattmahoney. [Online]. Available: http://www.mattmahoney.net/dc/textdata.html
[17] K. R. Rosenbloom *et al*., "The UCSC genome browser database: 2015 update," *Nucleic Acids Research.* vol. 43, pp. 670-681, Jan 2015.
[18] Apache Spark. Spark programming guide. [Online]. Available: http://spark.apache.org/docs/latest/programming-guide.html

**Kunaphas Kongkitimanon** was born in Bangkok, Thailand, in Nov. 1991. He received a bachelor of science degree in information and communication technology from the Faculty of Information and Communication Technology, Mahidol University, Bangkok, Thailand in 2013. Currently, he is a research assistant in the Integrative Computational BioScience Center, Mahidol University, Bangkok, Thailand and pursuing a master of science degree in computer science from the Faculty of Information and Communication Technology, Mahidol University.

**Boonsit Yimwadsana** was born in Bangkok, Thailand. He received his bachelor, master and Ph.D. degree in electrical engineering from Columbia University, New York, NY, USA in 2000, 2001 and 2007 respectively. Currently, he is working as a full-time faculty member at the Faculty of Information and Communication Technology, Mahidol University, Bangkok, Thailand and a principal investigator and member of the Integrative Computational Bioscience Center, Mahidol University, Bangkok, Thailand. Asst. Prof. Boonsit Yimwadsana is a member of IEEE since 1998.