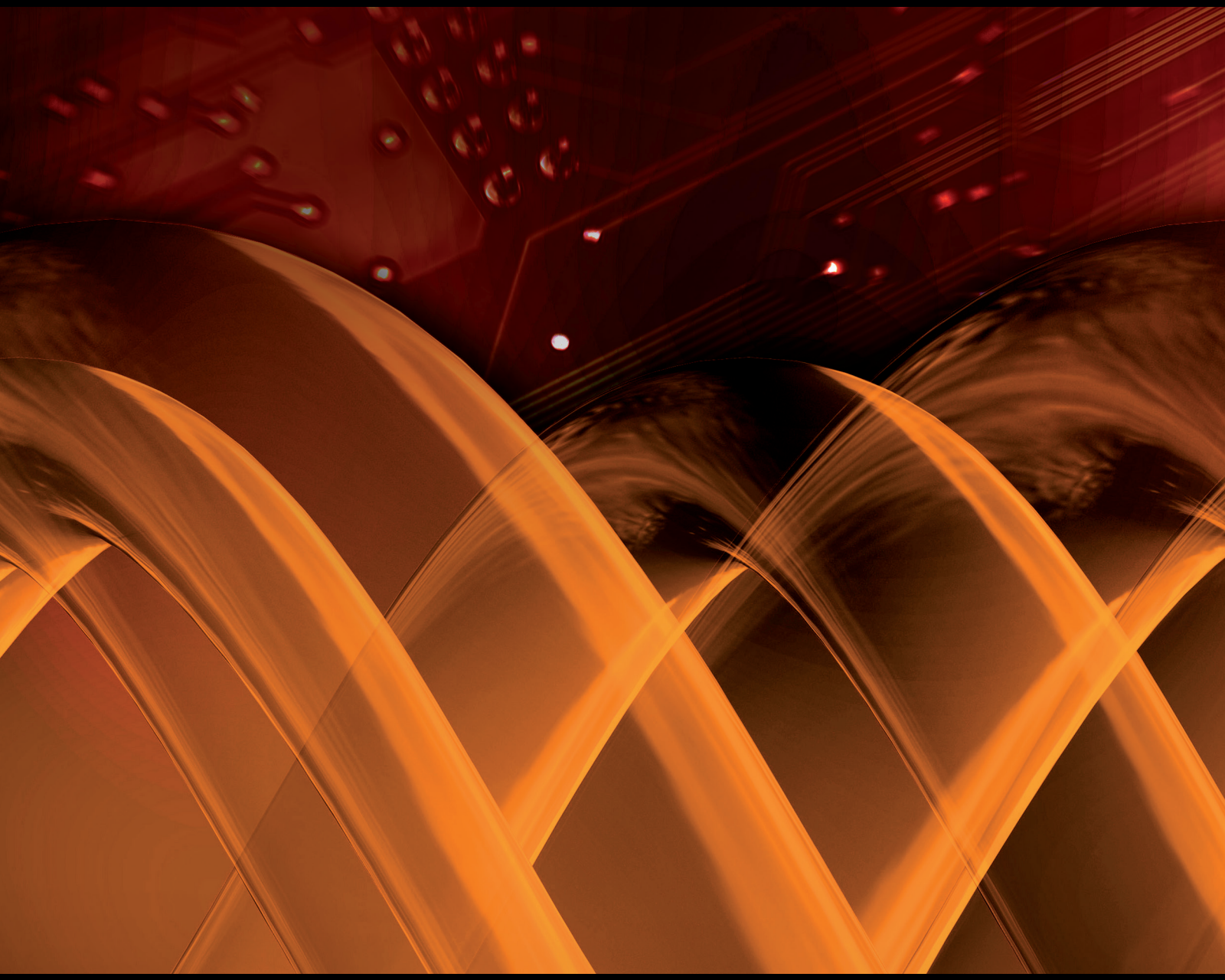
International Journal of Genomics

Novel Computational Technologies for Next-Generation Sequencing Data Analysis and Their Applications

Guest Editors: Chuan Yi Tang, Che-Lun Hung, Huiru Zheng, Chun-Yuan Lin, and Hai Jiang



Novel Computational Technologies for Next-Generation Sequencing Data Analysis and Their Applications

International Journal of Genomics

Novel Computational Technologies for Next-Generation Sequencing Data Analysis and Their Applications

Guest Editors: Chuan Yi Tang, Che-Lun Hung, Huiru Zheng, Chun-Yuan Lin, and Hai Jiang

Editorial Board

Contents

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 254685, 3 pages
http://dx.doi.org/10.1155/2015/254685

Editorial

Novel Computational Technologies for Next-Generation Sequencing Data Analysis and Their Applications

Chuan Yi Tang,[1] Che-Lun Hung,[2] Huiru Zheng,[3] Chun-Yuan Lin,[4] and Hai Jiang[5]

[1]*Department of Computer Science and Information Engineering, Providence University, 200 Sec. 7, Taiwan Boulevard, Shalu District, Taichung City 43301, Taiwan*
[2]*Department of Computer Science and Communication Engineering, Providence University, 200 Sec. 7, Taiwan Boulevard, Shalu District, Taichung City 43301, Taiwan*
[3]*School of Computing and Mathematics, Computer Science Research Institute, University of Ulster, Jordanstown Campus, Shore Road, Newtownabbey, County Antrim BT37 0QB, UK*
[4]*Department of Computer Science and Information Engineering, Chang Gung University, No. 259, Wenhua 1st Road, Guishan District, Taoyuan City 33302, Taiwan*
[5]*Department of Computer Science, Arkansas State University, 2105 Aggie Road, Jonesboro, AR 72401, USA*

Correspondence should be addressed to Che-Lun Hung; clhung@pu.edu.tw

Received 20 September 2015; Accepted 29 September 2015

1. Introduction

Next-generation sequencing (NGS) technologies, such as Illumina/Solexa, ABI/SOLiD, and Roche/454 Pyrosequencing, are revolutionizing the acquisition of genomic data at relatively low cost. NGS technologies are rapidly changing the approach to complex genomic studies, opening a way to the development of personalized drugs and personalized medicine. NGS technologies use massive throughput sequencing to obtain relatively short reads. NGS technologies will generate enormous datasets, in which even small genomic projects may generate terabytes of data. Therefore, new computational methods are needed to analyze a wide range of genetic information and to assist data interpretation and downstream applications, including high-throughput polymorphism detections, comparative genomics, prediction of gene function and protein structure, transcriptome analysis, mutation detection and confirmation, genome mapping, and drug design. The creation of large-scale datasets now poses a great computational challenge. It will be imperative to improve software pipelines, so that we can analyze genome data more efficiently.

Until now, many new computational methods have been proposed to cope with the big biological data, especially NGS sequence data. Also, many successful bioinformatics applications with NGS data through these methods have unveiled a lot of scientific results, which encourage biologists to adopt novel computing technologies. The research papers selected for this special issue represent recent progress in the aspects, including theoretical studies, novel algorithms, high performance computing technologies, and method and algorithm improvement. All of these papers not only provide novel ideas and state-of-the-art technologies in the field but also stimulate future research for next-generation sequencing data analysis and their applications.

2. Computational Genomics

Development of efficient algorithms for processing short nucleotide sequences has played a key role in enabling the uptake of DNA sequencing technologies in life sciences. In particular, reassembly of human genomes (*de novo* or reference guided) from short DNA sequence reads has had a substantial impact on health research. *De novo* assembly of the genome of a species is essential in the absence of a reference genome sequence. The paper by I. Birol et al. entitled "Spaced Seed Data Structures for *De Novo* Assembly"

introduces the data structure designs for spaced seeds in the form of paired k-mers to solve the limitation of the de Bruijn graph (DBG) paradigm for long reads. First, Bloom filter is used to store spaced seeds, and it can be tolerant of sequencing errors. Then they used a data structure for tracking the frequencies of observed spaced seeds.

Next-generation sequencing technologies are now producing multiple times the genome size in total reads from a single experiment. This is enough information to reconstruct at least some of the differences between the individual genome studied in the experiment and the reference genome of the species. However, in most typical protocols this information is disregarded and the reference genome is used. The paper by K. Buza et al. entitled "RECORD: Reference-Assisted Genome Assembly for Closely Related Genomes" proposes a new approach that allows researchers to reconstruct genomes very closely related to the reference genome (e.g., mutants of the same species) directly from the reads used in the experiment. Their approach applies *de novo* assembly software, called "RECORD," to experimental reads and so called pseudoreads and uses the resulting contigs to generate a modified reference sequence. In this way, it can very quickly, and at no additional sequencing cost, generate new, modified reference sequence that is closer to the actual sequenced genome and has a full coverage.

3. Metagenomics

Characterizing the taxonomic diversity for the planet-size data plays an important role in the metagenomic studies, while a crucial step for doing the study is the binning process to group sequence reads from similar species or taxonomic classes. The metagenomic binning remains a challenge work because of not only the various read noises but also the tremendous data volume. The paper by Y.-C. Lin entitled "A New Binning Method for Metagenomics by One-Dimensional Cellular Automata" introduces an unsupervised binning method for NGS reads based on the one-dimensional cellular automaton (1D-CA). The proposed method facilitates reducing the memory usage because 1D-CA costs only linear space.

4. High Performance Computing

The Smith-Waterman (SW) algorithm has been widely utilized for searching biological sequence databases in bioinformatics. However, the SW is a time-consuming algorithm and its usage may be limited by the sequence length and the number of sequences in a database. The previous works related to SW on GPGPU cannot solve the protein database search problem for the next-generation sequencing applications well. The paper by Y. Liu et al. entitled "Accelerating Smith-Waterman Alignment for Protein Database Search Using Frequency Distance Filtration Scheme Based on CPU-GPU Collaborative System" proposes an efficient SW alignment method, called CUDA-SWfr, for the protein database search by using the intratask parallelization technique based on a CPU-GPU collaborative system. Before doing the SW computations on GPU, a procedure is applied on CPU by using the frequency distance filtration scheme (FDFS) to eliminate the unnecessary alignments.

Compound comparison is an important task for the computational chemistry. By the comparison results, potential inhibitors can be found and then used for the pharmacy experiments. The time complexity of a pairwise compound comparison is $O(n^2)$, where n is the maximal length of compounds. The intrinsic time complexity of multiple compound comparison problem is $O(k^2n^2)$ with k compounds of maximal length n. The paper by C.-Y. Lin et al. entitled "Accelerating Multiple Compound Comparison Using LINGO-Based Load-Balancing Strategies on Multi-GPUs" proposes a GPU-based algorithm for MCC problem, called CUDA-MCC, on single- and multi-GPUs. Four LINGO-based load-balancing strategies are considered in CUDA-MCC in order to accelerate the computation speed among thread blocks on GPUs.

5. Genomics

The Circular Pattern Matching (CPM) problem appears as an interesting problem in many biological contexts. CPM consists in finding all occurrences of the rotations of a pattern P of length m in a text T of length n. The paper by Md. A. R. Azim et al. entitled "SimpLiFiCPM: A Simple and Lightweight Filter-Based Algorithm for Circular Pattern Matching" presents SimpLiFiCPM, a simple and lightweight filter-based algorithm to solve CPM problem. Much of the speed of the proposed algorithm comes from the fact that our filters are effective but extremely simple and lightweight.

Rapid advances in high-throughput sequencing techniques have created interesting computational challenges in bioinformatics. One of them refers to management of massive amounts of data generated by automatic sequencers. Therefore, an efficient data model to deal with a very large amount of nonconventional data, especially for writing and retrieving operations, is very important. The paper by R. Aniceto et al. entitled "Evaluating the Cassandra NoSQL Database Approach for Genomic Data Persistency" discusses the Cassandra NoSQL database approach for storing genomic data. They perform an analysis of persistency and I/O operations with real data, using the Cassandra database system.

6. Conclusion

All of the above papers address either novel computational strategies or applications for next-generation sequencing data. They also develop related method and approach improvements in applications of computational genomics and database. Honorably, this special issue serves as a landmark source for education, information, and reference to professors, researchers, and graduate students interested in updating their knowledge about or active in next-generation sequencing data analysis, metagenomics, computational genomics, and database system.

Acknowledgments

The guest editors would like to express sincere gratitude to numerous reviewers for their professional effort, insight, and hard work put into commenting on the selected articles which

reflect the essence of this special issue. They are grateful to all authors for their contributions and for undertaking two-cycle revisions of their manuscripts, without which this special issue could not have been produced.

Chuan Yi Tang
Che-Lun Hung
Huiru Zheng
Chun-Yuan Lin
Hai Jiang

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 502795, 7 pages
http://dx.doi.org/10.1155/2015/502795

Research Article

Evaluating the Cassandra NoSQL Database Approach for Genomic Data Persistency

Rodrigo Aniceto,[1] Rene Xavier,[1] Valeria Guimarães,[1] Fernanda Hondo,[1] Maristela Holanda,[1] Maria Emilia Walter,[1] and Sérgio Lifschitz[2]

[1]*Computer Science Department, University of Brasilia (UNB), 70910-900 Brasilia, DF, Brazil*
[2]*Informatics Department, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), 22451-900 Rio de Janeiro, RJ, Brazil*

Correspondence should be addressed to Maristela Holanda; mholanda@cic.unb.br

Received 11 March 2015; Accepted 14 May 2015

Academic Editor: Che-Lun Hung

Rapid advances in high-throughput sequencing techniques have created interesting computational challenges in bioinformatics. One of them refers to management of massive amounts of data generated by automatic sequencers. We need to deal with the persistency of genomic data, particularly storing and analyzing these large-scale processed data. To find an alternative to the frequently considered relational database model becomes a compelling task. Other data models may be more effective when dealing with a very large amount of nonconventional data, especially for writing and retrieving operations. In this paper, we discuss the Cassandra NoSQL database approach for storing genomic data. We perform an analysis of persistency and I/O operations with real data, using the Cassandra database system. We also compare the results obtained with a classical relational database system and another NoSQL database approach, MongoDB.

1. Introduction

Advanced hardware and software technologies increase the speed and efficiency with which scientific workflows may be performed. Scientists may execute a given workflow many times, comparing results from these executions and providing greater accuracy in data analysis. However, handling large volumes of data produced by distinct program executions under varied conditions becomes increasingly difficult. These massive amounts of data must be stored and treated in order to support current genomic research [1–4]. Therefore, one of the main problems when working with genomic data refers to the storage and search of these data, requiring many computational resources.

In computational environments with large amounts of possibly unconventional data, NoSQL [5] database systems have emerged as an alternative to traditional Relational Database Management Systems (RDBMS). NoSQL systems are distributed databases built to meet the demands of high scalability and fault tolerance in the management and analysis of massive amounts of data. NoSQL databases are coded in many distinct programming languages and are generally available as open-source software.

The objective of this paper is to study the persistency of genomic data on a particular and widely used NoSQL database system, namely, Cassandra [6]. The tests performed for this study use real genomic data to evaluate insertion and extraction operations into and from the Cassandra database. Considering the large amounts of data in current genome projects, we are particularly concerned with high performances. We discuss and compare our results with a relational system (PostgreSQL) and another NoSQL database system, MongoDB [7].

This paper is organized as follows. Section 2 presents a brief introduction for NoSQL databases and the main features of Cassandra database system. We discuss some related work in Section 3 and we present, at Section 4, the architecture of the database system. Section 5 discusses the practical results obtained and Section 6 concludes and suggests future works.

2. NoSQL Databases: An Overview

Many relevant innovations in data management came from Web 2.0 applications. However, the techniques and tools available in relational systems may, sometimes, limit their deployment. Therefore, some researchers have decided to develop their own web-scale database solutions [8].

NoSQL (not-only SQL) databases have emerged as a solution to storage scalability issues, parallelism, and management of large volumes of unstructured data. In general, NoSQL systems have the following characteristics [8–10]: (i) they are based on a nonrelational data model; (ii) they rely on distributed processing; (iii) high availability and scalability are main concerns; and (iv) some are schemaless and have the ability to handle both structured and unstructured data.

There are four main categories of NoSQL databases [8, 11–13]:

(i) Key-value stores: data is stored as key-pairs values. These systems are similar to dictionaries, where data is addressed by a single key. Values are isolated and independent from another, and relationships are handled by the application logic.

(ii) Column family database: it defines the data structure as a predefined set of columns. The *super columns* and *column family* structures can be considered the database schema.

(iii) Document-based storage: a document store uses the concept of key-value store. The documents are collections of attributes and values, where an attribute can be multivalued. Each document contains an ID key, which is unique within a collection and identifies document.

(iv) Graph databases: graphs are used to represent schemas. A graph database works with three abstractions: node, relationships between nodes, and key-value pairs that can attach to nodes and relationships.

2.1. Cassandra Database System. Cassandra is a cloud-oriented database system, massively scalable, designed to store a large amount of data from multiple servers, while providing high availability and consistent data [6]. It is based on the architecture of Amazon's Dynamo [14] and also on Google's BigTable data model [15]. Cassandra enables queries as in a key-value model, where each row has a unique row key, a feature adopted from Dynamo [6, 14, 16, 17]. Cassandra is considered a hybrid NoSQL database, using characteristics of both key-value and column oriented databases.

Cassandra's architecture is made of nodes, clusters, data centers and a *partitioner*. A node is a physical instance of Cassandra. Cassandra does not use a master-slave architecture; rather, Cassandra uses peer-to-peer architecture, which all nodes are equal. A cluster is a group of nodes or even a single node. A group of clusters is a data center. A *partitioner* is a hash function for computing the token of each row key.

When one row is inserted, a token is calculated, based on its unique row key. This token determines in what node that particular row will be stored. Each node of a cluster is responsible for a range of data based on a token. When the row is inserted and its token is calculated, this row is stored on a node responsible for this token. The advantage here is that multiple rows can be written in parallel into the database, as each node is responsible for its own write requests. However this may be seen as a drawback regarding data extraction, becoming a bottleneck. The *MurMur3Partitioner* [17] is a partitioner that uses tokens to assign equal portions of data to each node. This technique was selected because it provides fast hashing, and its hash function helps to evenly distribute data to all the nodes of a cluster.

The main elements of Cassandra are *keyspaces*, column families, columns, and rows [18]. A *keyspace* contains the processing steps of the data replication and is similar to a schema in a relational database. Typically, a cluster has one *keyspace* per application. A column family is a set of key-value pairs containing a column with its unique row keys. A column is the smallest increment of data, which contains a name, a value, and a timestamp. Rows are columns with the same primary key.

When a write operation occurs, Cassandra immediately stores the instruction on the Commit log, which goes into the hard disk (HD). Data from this write operation is stored at the *memtable*, which stays in RAM. Only when a predefined memory limit is reached, this data is written on *SSTables* that stay in the HD. Then, the Commit log and the *memtable* are cleaned up [18, 19]. In case of failure regarding the *memtables*, Cassandra reexecutes the written instructions available at the Commit log [19, 20].

When an extract instruction is executed, Cassandra first searches information in *memtables*. A large RAM allows large amounts of data in *memtables* and less data in HD, resulting in quick access to information [16].

3. Storing Genomic Data

Persistency of genomic data is not a recent problem. In 2004, Bloom and Sharpe [21] described the difficulties of managing these data. One of the main difficulties was the growing number of data generated by the queries. The work in Röhm and Blakeley [22] and Huacarpuma [23] consider relational databases (SQL Server 2008 and PostgreSQL, resp.) to store genomic data in FASTQ format.

Bateman and Wood [24] have suggested using NoSQL databases as a good alternative to persisting genetic data. However, no practical results are given. Ye and Li [25] proposed the use of Cassandra as a storage system. They consider multiple nodes so that there were no gaps in the consistency of the data. Wang and Tang [26] indicated some instructions for creating an application to perform data operations in Cassandra.

Tudorica and Bucur [27] compared some NoSQL databases to a MySQL relational database using the YCSB (Yahoo! Cloud Serving Benchmark). They conclude that in an environment where write operations prevail MySQL has a significantly higher latency when compared to Cassandra. Similar results about performance improvements for writing operations in Cassandra, when compared to MS SQL Express, were also reported by Li and Manoharan [28].

Many research works [25–28] present results involving the performance of a Cassandra database system for massive data volumes. In this paper, we have decided to evaluate the performance of Cassandra NoSQL database system specifically for genomic data.

4. Case Study

To validate our case study we have used real data. The sequences (also called *reads*) were obtained from liver and kidney tissue samples of one human male from the SRA-NCBI (http://trace.ncbi.nlm.nih.gov/Traces/sra/sra.cgi?), sequenced by the Illumina Genome Analyzer. It produced 72,987,691 sequences for the kidney samples and 72,126,823 sequences for the liver samples, each sequence containing 36 bases. Marioni et al. [29] generated these sequences.

FASTQ file stores sequences of nucleotides and their corresponding quality values. Three files were obtained from filtered sequences sampled from kidney cells, and another three files consisted of filtered genomic sequences sampled from liver cells. It should be noted that these data were selected because they were in FASTQ [1] format, which is commonly used in bioinformatics workflows.

In this case study, we carried out three analyses. In the first one, we investigated how Cassandra behaves when the computational environment is composed of a cluster with two and four computers. In the second one, we analyze the behavior of Cassandra compared to PostgreSQL, a relational database. In the last case study, we used the MongoDB document-oriented NoSQL database to compare to Cassandra's results.

4.1. Cloud Environment Architecture. In order to investigate the expected advantages of Cassandra's scalability, we have created two cloud environments: one with two nodes and the other with four nodes. Cassandra was installed on every node of the cluster. We have also used *OpsCenter* 4.0 [30], a DSE tool that implements a browser-based interface to remotely manage the cluster configuration and architecture. The architecture contains a single data center, named DC1. A single cluster, named *BIOCluster*, containing the nodes, was created, working with DC1.

4.2. Java Client. At the software level, we have defined the following functional requirements: (i) create a keyspace; (ii) create a table to store a FASTQ file; (iii) create a table with the names of inserted FASTQ files and their corresponding metadata; (iv) receive an input file containing data from a FASTQ file and insert it into a previously created table, followed by the file name and metadata; (v) extract all data from a table containing the contents of a FASTQ file; and (vi) remove the table and the keyspace.

Nonfunctional requirements were also defined: (i) the use of Java API, provided by DataStax, in order to have a better integration between the Cassandra distribution and the developed client application; (ii) the use of Cassandra Query Language (CQL3) [17], for database interactions, which is the current query language of Cassandra and resembles SQL; (iii) conversion to JSON files to be used by the client application, since it is simpler to work with JSON files in Java; and (iv) a good performance in operations.

With respect to this last requirement, three applications were developed, two for data conversion and one client application for Cassandra.

(1) FastqTojson Application converts the FASTQ input file into smaller JSON files, each JSON file with five hundred thousand reads. The objective is to load these small JSON files because, usually, FASTQ file occupies a few gigabytes. Furthermore, as it presents a proper format for the Java client, it does not consume many computational resources. Each JSON file occupies ten thousand rows in the database: each row is an array of ten columns; each field value of the column contains five *reads*.

(2) Cassandra client was also developed in Java, using the Java API provided by DataStax and is the one in which the data persists. This client creates a keyspace, inserts all JSON files from the first application in a single table, and extracts the data from a table.

For the database schema, it consists of a single *keyspace*, called *biodata*, a single cluster, called *biocluster*, one table of metadata and one table for each file persisting, as shown in Figure 1.

The allocation strategy for replicas and the replication factor are properties from the *keyspace*. The allocation strategy determines whether or not data is distributed through a network of different clusters. The Simple Strategy [31] was selected since this case study was performed in a single cluster. Likewise, since we did not consider failures and our goal was to study performance rather than fault recovery, we have chosen one replication factor. It should be noted that the replication factor determines the number of replicas distributed along the cluster. Focusing on performance, a higher number of replicas would also interfere on the insertion time.

As previously mentioned, the client application creates a table for each inserted FASTQ file, which has the same name of the file. Each of these tables has eleven columns, and each cell stores a small part of a JSON file, ten *reads* per cell, which is about 1 MB in size. This small set for columns and cells is due to the efficiency of Cassandra when a small number of columns are used and a big number of rows. This is also a consequence of the ability of *MurMur3Partitioner* to distribute each row in one node. Therefore, the cluster has a better load balance during insertions and extractions.

Once a table is created, the client inserts all data from JSON in the first stage on the database, as shown in Figure 2. In what follows, a single row is inserted into the metadata table containing as a row key the name of FASTQ file and a column with the number of rows. This latter is inserted into the metadata table to solve the memory limit of the Java Virtual Machine, which may happen when querying large tables.

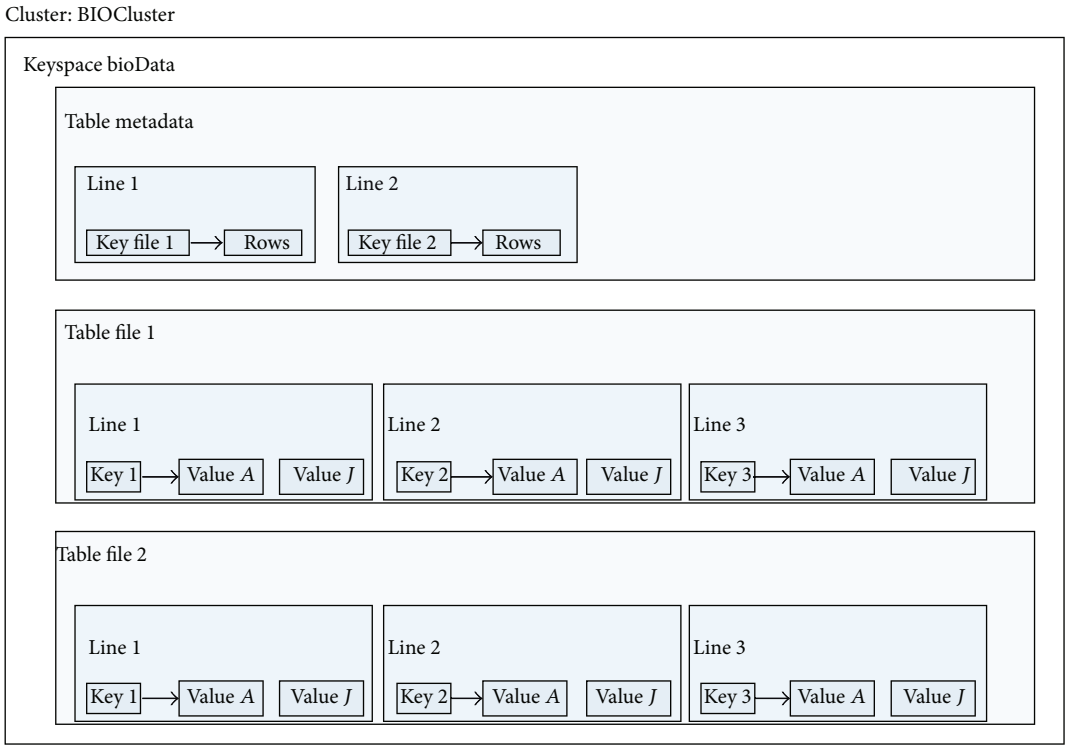


Figure 1: Database schema.

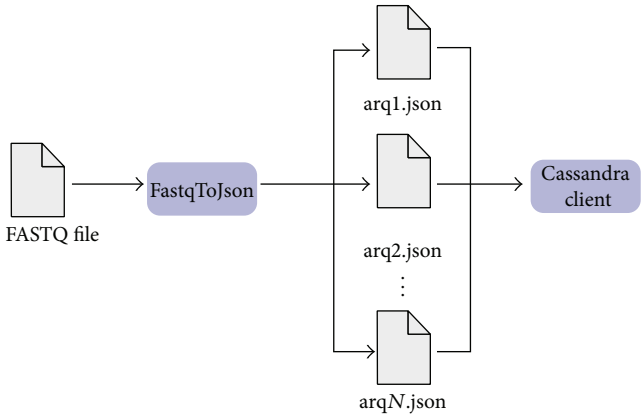


Figure 2: Stages of insertion.

When extracting data, the client queries the metadata table to get the number of rows on the table with the FASTQ data and then proceeds to the table extraction, which is done row by row and written into an ".out" file.

(3) OutToJso Application. After data extraction, there is a single file with the extension ".out." This application converts this file into a FASTQ format, making it identical to the original input file, resulting only in the FASTQ file without temporary file ".out." This process is shown in Figure 3.

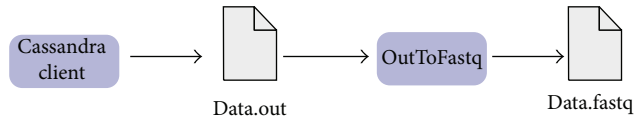


Figure 3: Stages of extraction.

5. Results

In this work, we have considered three experimental case studies to evaluate data consistency and performance for storing and extracting genomic data. For the first one, we verified Cassandra's scalability and variation in performance. For the second case study, we compared the Cassandra results to a PostgreSQL relational system and, finally, we used the MongoDB NoSQL database and compared other results to Cassandra NoSQL system. The case studies used the same data to insert and read sequences.

During the Cassandra evaluation, we have created two clusters. The first one, a Cassandra cluster with two computers, was created, while for the second one, a new cluster with four computers was created. The first cluster consisted of two computers with Intel Xeon E3-1220/3.1 GHz processor, one with 8 GB RAM and the other with 6 GB RAM. For the second cluster, besides the same two computers, two other computers with Intel Core i7 processor and 4 GB RAM was included. Each one of them used Ubuntu 12.04.

5.1. Insertions and Extractions Cassandra NoSQL. The input files are six FASTQ files with filtered data from kidney and liver cells. Table 1 shows the sizes of the file and the number

Table 1: Cells files.

File	File number	Size	Number of lines
Liver cells files	1	9,0 GB	850.933
	2	4,0 GB	358.841
	3	3,2 GB	286.563
Kidney cells files	4	6,9 GB	648.612
	5	3,8 GB	335.973
	6	5,3 GB	475.210

of rows that their respective JSON file had when inserted into Cassandra.

We have based the performance analyses on the elapsed time to store (insert) data into and to retrieve (extract) data from the database. These elapsed times are important because if one wants to use the Cassandra system in bioinformatics workflows, it is necessary to know how long the data becomes available to execute each program.

Table 2 shows the elapsed times to insert and extract sequences in the database, with both implementations. Columns 3 and 5 show the insertions using two nodes. Similarly, columns 4 and 6 show the extractions using four nodes. As expected, we could confirm the hypothesis that the database performance increases when we add more nodes.

Figures 4 and 5 show comparative charts of insertion and extraction elapsed times according to the number of computers that Cassandra considers. Insertion into two computers is longer than using four computers. Here the performance also improves when the number of computers increases in the cluster.

5.2. Comparison of Relational and Cassandra NoSQL Systems. We compared the Cassandra results with Huacarpuma [23] that used the same data to insert and read sequences in the PostgreSQL, a relational database. In the latter experiment, the author used only one server with an Intel Xeon processor, eight cores of 2.13 GHz and 32 GB RAM, executing Linux Server Ubuntu/Linaro 4.4.4-14.

The server's RAM for the relational database is larger than the sum of the memories of the four computers used in this experiment. Nonetheless, we use the results of the relational database to demonstrate that it is possible to achieve high performances even with a modest hardware due to scalability and parallelism.

Table 3 shows the sum of the insertion and extraction times in the relational database and the two computational environments using Cassandra, Cassandra (2), a cluster with two computers, and Cassandra (4), a cluster with four computers.

The writing time in Cassandra is lower due to parallelism, as seen in Table 3. Write actions in Cassandra are more effective than in a relational database. However, its performance was lower for query answering, as shown in Figure 6. This is due to two factors: first, Cassandra had to ensure that the returned content was in its latest version, verifying the data divided between machines; second, the data size is larger than the available RAM; therefore, part of the data had to be stored in SSTable, reducing the speed of the search.

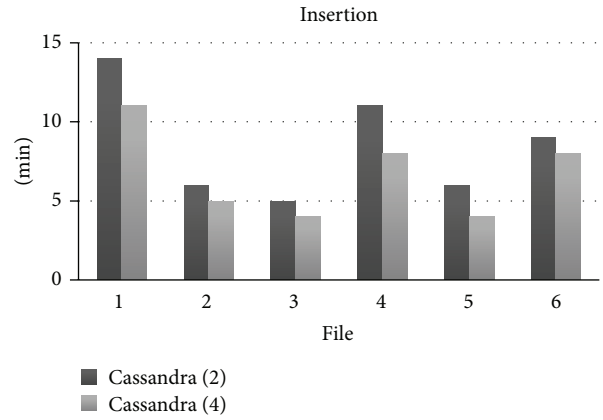


Figure 4: Comparison between inserts (time × file number).

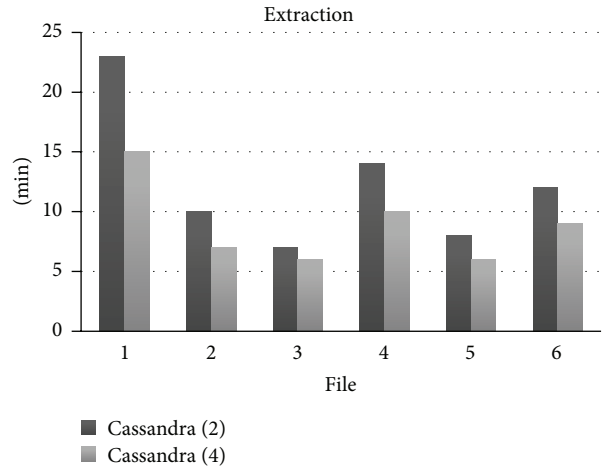


Figure 5: Comparison between extractions (time × file number).

The reader should note that the results obtained with Cassandra just indicate a trend. They are not conclusive because the hardware characteristics of all experiments are different.

Nevertheless, the improved performance with the increase of nodes is an indication that Cassandra may sometimes surpass relational database systems in a larger number of computers, making its use viable in data searches in bioinformatics.

5.3. Comparison of MongoDB and Cassandra NoSQL Databases. We compared the Cassandra results to the same data to insert and read sequences in a MongoDB NoSQL. This is an open-source document-oriented NoSQL database designed to store large amounts of data.

The server where we have installed MongoDB is an i7 processor with 16 GB RAM. This server has 2 GB RAM more. The server where we have installed MongoDB had 2 GB RAM more than cluster with two computers, Cassandra (2), and 6 GB RAM less than the sum of the RAM memories of four computers, Cassandra (4).

Table 2: Times to insert and extract sequences from the database.

File	Size	Insertion		Extraction	
		Cassandra (2)	Cassandra (4)	Cassandra (2)	Cassandra (4)
1	9,0 GB	14 m 30 s 645 ms	11 m 44 s 105 ms	23 m 37 s 964 ms	15 m 04 s 158 ms
2	4,0 GB	6 m 10 s 471 ms	05 m 05 s 710 ms	9 m 41 s 018 ms	7 m 34 s 523 ms
3	3,2 GB	5 m 05 s 914 ms	4 m 51 s 823 ms	7 m 39 s 188 ms	6 m 02 s 648 ms
4	6,9 GB	11 m 25 s 899 ms	8 m 27 s 630 ms	14 m 25 s 120 ms	10 m 00 s 031 ms
5	3,8 GB	6 m 09 s 417 ms	4 m 42 s 386 ms	8 m 37 s 890 ms	6 m 05 s 487 ms
6	5,3 GB	8 m 43 s 330 ms	8 m 05 s 215 ms	12 m 23 s 855 ms	9 m 03 s 041 ms

Table 3: PostgreSQL and Cassandra results.

Database	Insertion	Extraction
PostgreSQL	1 h 51 m 54 s	28 m 27 s
Cassandra (2)	52 m 5 s	1 h 16 m 25 s
Cassandra (4)	42 m 56 s	53 m 49 s

Table 4: MongoDB and Cassandra final results.

Database	Insertion	Extraction
MongoDB	45 m 17 s	19 m 13 s
Cassandra (2)	52 m 5 s	1 h 16 m 25 s
Cassandra (4)	42 m 56 s	53 m 49 s

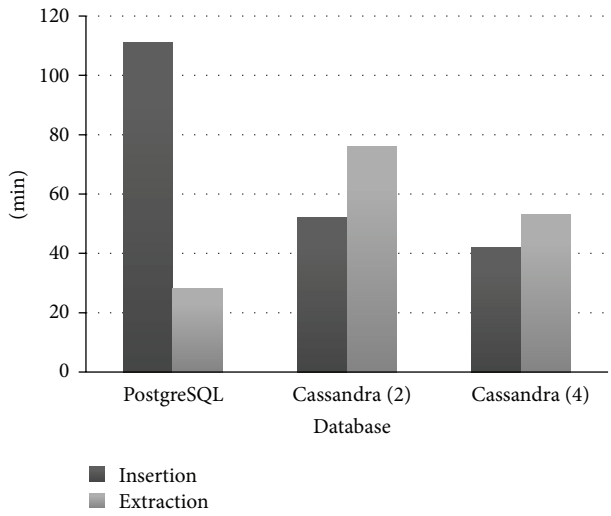


Figure 6: Comparison between Cassandra and PostgreSQL.

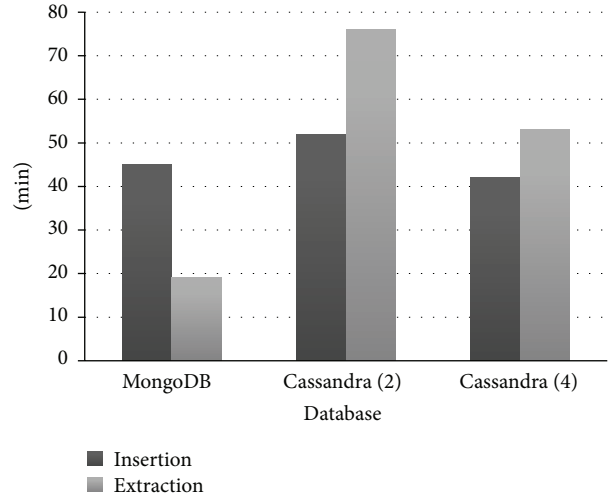


Figure 7: Comparison between Cassandra and MongoDB database.

Table 4 shows the sum of the insertion and extraction times in the MongoDB database and the Cassandra with two and four computers in a cluster. The performances of insertion operations were similar using either MongoDB or Cassandra databases. However, the MongoDB showed better behavior than Cassandra NoSQL in the extraction of genomic data in FASTQ format.

In Figure 7 our results suggest that there is a similar behavior of the insertions in both MongoDB and Cassandra. There was a performance gain of more than 50% in the extraction, when comparing the results of a Cassandra in a cluster with two computers and another cluster with four computers.

6. Conclusions

In this work we studied genomic data persistence, with the implementation of a NoSQL database using Cassandra. We have observed that it presented a high performance for writing operations due to the larger number of massive insertions compared to data extractions. We used the DSE tool together with Cassandra, which allowed us to create a cluster and a client application suitable for the expected data manipulation.

Our results suggest that there is a reduction of the insertion and query times when more nodes are added in Cassandra. There was a performance gain of about 17% in the insertions and a gain of 25% in reading, when comparing the results of a cluster with two computers and another cluster with four computers.

Comparing the performance of Cassandra to the MongoDB database, the results of MongoDB indicate that the extraction of the MongoDB is better than Cassandra. For data insertions the behaviors of Cassandra and MongoDB were similar.

From the results presented here, it is possible to outline new approaches in studies of persistency regarding genomic

data. Positive results could boost new research, for example, the creation of a similar application using other NoSQL databases or new tests using Cassandra with different hardware configurations seeking improvements in performance. It is also possible to create a relational database with hardware settings identical to Cassandra, in order to make more detailed comparisons.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

References

[1] S. A. Simon, J. Zhai, R. S. Nandety et al., "Short-read sequencing technologies for transcriptional analyses," *Annual Review of Plant Biology*, vol. 60, no. 1, pp. 305–333, 2009.

[2] M. L. Metzker, "Sequencing technologies—the next generation," *Nature Reviews Genetics*, vol. 11, no. 1, pp. 31–46, 2010.

[3] C.-L. Hung and G.-J. Hua, "Local alignment tool based on Hadoop framework and GPU architecture," *BioMed Research International*, vol. 2014, Article ID 541490, 7 pages, 2014.

[4] Y.-C. Lin, C.-S. Yu, and Y.-J. Lin, "Enabling large-scale biomedical analysis in the cloud," *BioMed Research International*, vol. 2013, Article ID 185679, 6 pages, 2013.

[5] K. Kaur and R. Rani, "Modeling and querying data in NoSQL databases," in *Proceedings of the IEEE International Conference on Big Data*, pp. 1–7, October 2013.

[6] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[7] K. Chodorow, *MongoDB—The definitive Guide*, O'Reilly, 2nd edition, 2013.

[8] R. Hecht and S. Jablonski, "NoSQL evaluation: a use case oriented survey," in *Proceedings of the International Conference on Cloud and Service Computing (CSC '11)*, pp. 336–341, December 2011.

[9] Y. Muhammad, *Evaluation and implementation of distributed NoSQL database for MMO gaming environment [M.S. thesis]*, Uppsala University, 2011.

[10] C. J. M. Tauro, S. Aravindh, and A. B. Shreeharsha, "Comparative study of the new generation, agile, scalable, high performance NOSQL databases," *International Journal of Computer Applications*, vol. 48, no. 20, pp. 1–4, 2012.

[11] R. P. Padhy, M. Patra, and S. C. Satapathy, "RDBMS to NoSQL: reviewing some next-generation non-relational databases," *International Journal of Advanced Engineering Science and Technologies*, vol. 11, no. 1, pp. 15–30, 2011.

[12] M. Bach and A. Werner, "Standardization of NoSQL database languages," in *Beyond Databases, Architectures, and Structures: 10th International Conference, BDAS 2014, Ustron, Poland, May 27–30, 2014. Proceedings*, vol. 424 of *Communications in Computer and Information Science*, pp. 50–60, Springer, Berlin, Germany, 2014.

[13] M. Indrawan-Santiago, "Database research: are we at a crossroad? Reflection on NoSQL," in *Proceedings of the 15th International Conference on Network-Based Information Systems (NBIS '12)*, pp. 45–51, IEEE, Melbourne, Australia, September 2012.

[14] G. DeCandia, D. Hastorun, M. Jampani et al., "Dynamo: amazon's highly available key-value store," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pp. 205–220, ACM, October 2007.

[15] F. Chang, J. Dean, S. Ghemawat et al., "Bigtable: a distributed storage system for structured data," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pp. 205–218, 2006.

[16] E. Hewitt, *Cassandra—The Definitive Guide*, O'Reilly, 1st edition, 2010.

[17] M. Klems, D. Bermbach, and R. Weinert, "A runtime quality measurement framework for cloud database service systems," in *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC '12)*, pp. 38–46, September 2012.

[18] V. Parthasarathy, *Learning Cassandra for Administrators*, Packt Publishing, Birmingham, UK, 2013.

[19] DataStax, Apache Cassandra 1.2 Documentation, 2015, http://www.datastax.com/documentation/cassandra/1.2/pdf/cassandra12.pdf.

[20] M. Fowler and P. J. Sadalage, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, Pearson Education, Essex, UK, 2014.

[21] T. Bloom and T. Sharpe, "Managing data from high-throughput genomic processing: a case study," in *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '04)*, pp. 1198–1201, 2004.

[22] U. Röhm and J. A. Blakeley, "Data management for high throughput genomics," in *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR '09)*, Asilomar, Calif, USA, January 2009, http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_31.pdf.

[23] R. C. Huacarpuma, *A data model for a pipeline of transcriptome high performance sequencing [M.S. thesis]*, University of Brasília, 2012.

[24] A. Bateman and M. Wood, "Cloud computing," *Bioinformatics*, vol. 25, no. 12, p. 1475, 2009.

[25] Z. Ye and S. Li, "A request skew aware heterogeneous distributed storage system based on Cassandra," in *Proceedings of the International Conference on Computer and Management (CAMAN '11)*, pp. 1–5, May 2011.

[26] G. Wang and J. Tang, "The NoSQL principles and basic application of cassandra model," in *Proceedings of the International Conference on Computer Science and Service System (CSSS '12)*, pp. 1332–1335, August 2012.

[27] B. G. Tudorica and C. Bucur, "A comparison between several NoSQL databases with comments and notes," in *Proceedings of the 10th RoEduNet International Conference on Networking in Education and Research (RoEduNet '11)*, pp. 1–5, June 2011.

[28] Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," in *Proceedings of the 14th IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM '13)*, pp. 15–19, August 2013.

[29] J. C. Marioni, C. E. Mason, S. M. Mane, M. Stephens, and Y. Gilad, "RNA-seq: an assessment of technical reproducibility and comparison with gene expression arrays," *Genome Research*, vol. 18, no. 9, pp. 1509–1517, 2008.

[30] *OpsCenter 4.0 User Guide Documentation*, DataStax, 2015, http://www.datastax.com/documentation/opscenter/4.0/pdf/opscuserguide40.pdf.

[31] DataStax, DataStax Enterprise 3.1 Documentation, 2015, http://www.datastax.com/doc-source/pdf/dse31.pdf.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 563482, 10 pages
http://dx.doi.org/10.1155/2015/563482

Research Article

RECORD: Reference-Assisted Genome Assembly for Closely Related Genomes

Krisztian Buza, Bartek Wilczynski, and Norbert Dojer

Faculty of Mathematics, Informatics and Mechanics (MIM), University of Warsaw, Banacha 2, 02-097 Warsaw, Poland

Correspondence should be addressed to Krisztian Buza; buza@biointelligence.hu

Received 18 March 2015; Revised 27 May 2015; Accepted 31 May 2015

Academic Editor: Chun-Yuan Lin

Background. Next-generation sequencing technologies are now producing multiple times the genome size in total reads from a single experiment. This is enough information to reconstruct at least some of the differences between the individual genome studied in the experiment and the reference genome of the species. However, in most typical protocols, this information is disregarded and the reference genome is used. *Results*. We provide a new approach that allows researchers to reconstruct genomes very closely related to the reference genome (e.g., mutants of the same species) directly from the reads used in the experiment. Our approach applies de novo assembly software to experimental reads and so-called pseudoreads and uses the resulting contigs to generate a modified reference sequence. In this way, it can very quickly, and at no additional sequencing cost, generate new, modified reference sequence that is closer to the actual sequenced genome and has a full coverage. In this paper, we describe our approach and test its implementation called RECORD. We evaluate RECORD on both simulated and real data. We made our software publicly available on sourceforge. *Conclusion*. Our tests show that on closely related sequences RECORD outperforms more general assisted-assembly software.

1. Background

The emergence of population genomic projects leads to an ever growing need for software and methods that facilitate studying closely related organism with next-generation sequencing technologies. This includes determination of the genomic sequences of individuals in the presence of the more generic reference genome of the species. This task is known as reference-assisted genome assembly and many ongoing research projects depend on the accurate solution for this problem.

In recent years, next-generation sequencing technologies have brought us the possibility to simultanously sequence millions of short DNA fragments in a DNA library prepared from almost any biochemical experiment [1]. Great improvement in the quality and amount of short reads obtained from a single experiment allowed for development of many more biochemical assays [2] such as MNase-seq [3], DNAse-seq [4], or Chia-Pet [5] in addition to the more standard ChIP-Seq [6] or RNA-seq [7]. Similarly, the next-generation sequencing techniques may be applied to metagenomic samples returning short reads originating from multiple genomes including some potentially unknown species.

Importantly, many of these techniques require the prior knowledge of the reference genome of the species for which the experiment was performed. This genome sequence is used to map the reads and obtain the final readout of the experiment as the read counts per base pair. Such procedures are guaranteed to work very well only under the assumption that we know the exact sequence of the genome under study. There are, however, many biologically relevant cases when this assumption cannot be satisfied. For example, in quickly growing cell populations such as cancer cell-lines or microbial colonies, even rare mutations can get fixed in the population very quickly. This leads to situations where sampled sequences can significantly differ from the original reference genome. Similarly, many lab experiments involve genetically modified cells or organisms. While these modifications are usually controlled as much as possible, the researchers frequently do not know the exact landing site of the introduced

sequence or the number of copies in which it was integrated into the host genome. This has naturally serious implications for the accuracy of the results because any difference between the reference genome and the sampled one will lead to differences in the expected number of reads mappable to the reference genome at the differing position. This in turn can interfere with the measurement of the real abundance of this DNA region in the sample.

This problem can be, at least theoretically, alleviated by introducing an additional step into the process: instead of directly mapping the reads to the reference genome, we can create a "modified" assembly of the genome based on the reads from the sample and the reference genome. Then, we can use this assembly to map the reads and measure their abundance. This approach can be broken down into two major steps:

(I) Assembling a genome of the sampled population based on the obtained reads and the reference genome.

(II) Assessing the abundance of reads in genomic regions using such an improved reference sequence.

In the early years of next-generation sequencing, the first step of such an approach was impractical, as the number of reads used for an experiment like ChIP-seq was far too low and their quality was not high enough to attempt assembly of a better reference genome than the one deposited in the databases by the relevant genome consortium. However, now it is commonplace that the total of sequencing reads generated for a single experiment such as ChIA-PET might be covering the genome multiple times and, at least in case of the model organisms such as *D. melanogaster* or *C. elegans*, the read lengths might be large enough to attempt an assembly.

This approach also has another limitation. If the reference sequence is very different from the one used in the experiment, it contributes more to a problem than to a solution. Any attempt to use a completely unrelated sequence as a reference in such an approach is bound to introduce errors. Therefore, in order to ensure that the output of the assembly is useful, when we provide a method of generating reference-assisted assemblies, it is crucial to validate that the reference is actually close enough to the target genome.

In this paper, we focus on developing an approach for reference-assisted genome assembly. We assume that the actual genome of the organism and the reference genome are close to each other; for example, the reference genome of the species under consideration is given, but not the genome of the particular mutant. We point out that currently used straightforward solutions produce suboptimal or, in some cases, even misleading results. For example, when simply assembling the genome from the given reads, due to the low coverage of those reads, we may obtain too short contigs leading to an assembly useless in practical applications. Consequently, we need an assembly technique which fulfills the following criteria. First of all, it should output sequences that are long enough even in cases when the coverage of the genome sequence by the experimental reads is relatively low. Second, not only should the output sequences be large enough individually, but, together, they should cover as much as possible of the genome in order to allow detection of the abundance of reads in any region of the genome. Third, the result of the assembly should be accurate; that is, the assembled genome should be as close as possible to the actual genome of the studied organism. Last, but not least, we aim to provide a *simple* assembly approach. With simplicity, we mean computational time (in order to keep the entire process computationally tractable) and the method clarity needed for ease of reproducibility and reuse of presented ideas in the context of different specific protocols. We believe the adaptation of the ideas presented in this paper may be straightforward in some applications, including RNA-Seq and ChIP-Seq. In other cases, it would be possible after substantial effort. For example, metagenomic sequencing might potentially benefit from some ideas presented in this paper. However, the currently presented approach would need to accommodate multiple genomes and reads originating from different, related species that may be present in the sample at the same time.

The growing interest in genome assembly is also reflected by recent publications. For example, Peng and Smith [8] studied genome assembly from the theoretical point of view and showed that various combinatorial problems related to genome assembly are NP-hard. On the other hand, various methods have been proposed for reference-assisted genome assembly, such as Amos [9], RACA [10], ARACHNE [11, 12], IMR/DENOM [13], RAGOUT [14], AlignGraph [15], and the pipeline developed by Gnerre et al. [16] which was developed inside the framework provided by ARACHNE. Similarly to our approach, Gnerre et al. used a de novo assembler as a component. They mapped reads to several reference genomes and used the resulting mapping information to improve the output of the de novo assembly in subsequent steps. In contrast to Gnerre et al., we only use one reference genome, and, more importantly, we use the reference genome to provide enriched *input* for the de novo assembler. Furthermore, we assume that the reference is closely related to the target genome, and therefore the reference is directly used to determine order and orientation of the assembly contigs. In contrast, RACA focused on reliable order and orientation of the contigs. Amos, one of the most popular assisted assembly softwares, aligns reads to the reference genome and uses alignment and layout information to generate a new consensus sequence [9]. We note that the techniques presented in this paper are orthogonal to the ones used in the aforementioned works; that is, as future work, RECORD may be combined with other assisted assembly tools. In this paper, we focus on experimentally evaluating the power of the relatively simple techniques of our pipeline. We will show that, despite their simplicity, they may achieve surprisingly good results.

In the next section we describe our approach, a simple but surprisingly effective reference-assisted assembly technique, and the software that implements it. By design, this approach is most useful in cases when the reference and target genomes are closely related, and the coverage of the target genome by the experimental reads is relatively low such as multiplexing scenarios where multiple experimental DNA libraries are barcoded and pooled in a single sequencing lane. Subsequently we present the results of the experimental evaluation

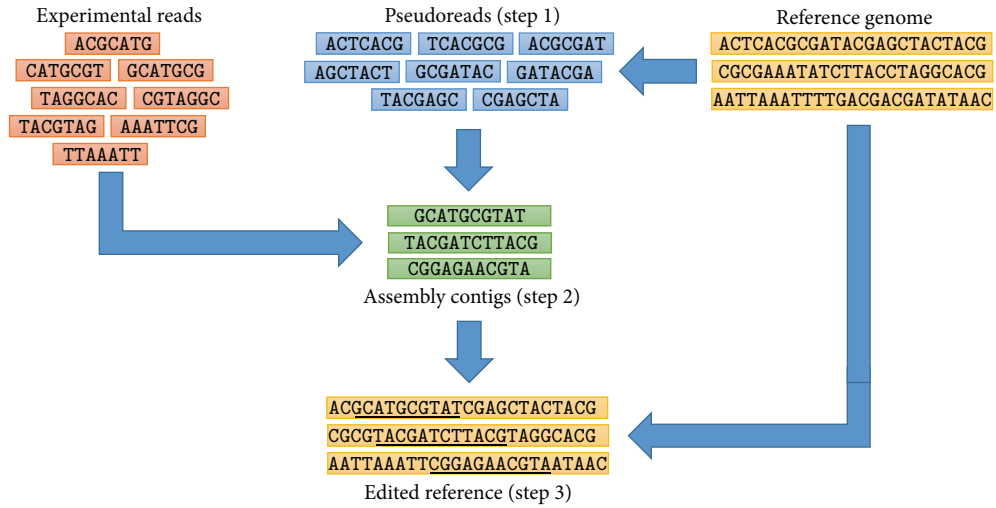


FIGURE 1: RECORD: Reference-Assisted Genome Assembly for Closely Related Genomes. The inputs of the pipeline, that is, the experimental reads and the reference genome, are illustrated in the top left and top right of the figure, respectively. Intermediate results produced in various steps of the analysis process are depicted. The dependency between these intermediate results is shown by arrows. In the illustration of the 3rd step, we underlined those segments of the edited reference which were replaced by one of the assembly contigs.

of RECORD and compare it to Amos [9], one of the most popular assisted assembly tools. We show that, under realistic conditions of approximately 1 percent divergence between reference genome and the studied sequence, our approach outperforms naive approaches and Amos (which excels in situations where the divergence is much higher). To ensure reproducibility and extensibility of our work, we evaluate our approach on several collections of publicly available next-generation sequencing data sets originating from various model organisms such as yeast (*S. pombe*), fruit fly (*D. melanogaster*), and plant (*A. thaliana*).

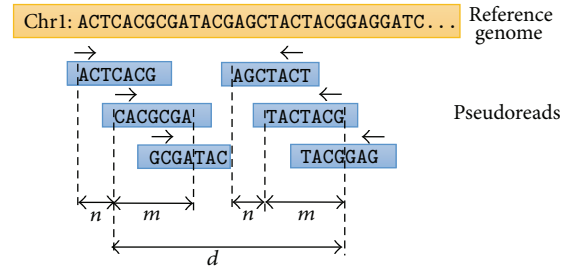


FIGURE 2: Generation of pseudoreads from the reference genome.

2. Implementation

We propose RECORD, Reference-Assisted Genome Assembly for Closely Related Genomes. Our approach consists of the following steps (see Figure 1):

(1) We generate pseudoreads from the reference genome. We generate pseudoreads in order to ensure that the coverage of the genome is large enough.

(2) We obtain the contigs of the actual genome of the organism using a genome assembler, such as Velvet [17]. As input of the assembler, we propose to use the pseudoreads generated in the previous step together with the experimental reads.

(3) We create an edited reference genome. The contigs obtained in the previous step may not cover the actual genome of the organism entirely, and, more importantly, the genome obtained in the previous steps may be fragmented into a relatively large amount of contigs. Therefore, the contigs obtained in the previous step will be mapped to the reference genome with MUMmer [18]. Using the reference genome and the mapped contigs, we produce a new genome, called edited reference, the segments of which are replaced according to the mapping. This step ensures that the edited reference is close to the true genome of the organism, while it covers as much regions of the genome as possible.

Below we give a detailed description of the above steps.

2.1. Generation of Pseudoreads from the Reference. While generating pseudoreads from the reference, we make sure that these pseudoreads have uniform coverage and large enough overlaps so that they can "assist" the genome assembler, while it joins reads to contigs. In particular, we generate reads of length m from each chromosome beginning at positions $0, n, 2{\cdot}n, \ldots, k{\cdot}n, \ldots$, where m and n are parameters that can be set by the user. We generate paired-end reads; the first mate of the paired-end reads is generated directly from the reference, while the second mate is generated from its reverse complement, so that the resulting data has similar character as the paired-end reads in NGS experiments. The distance between the ends of the mates of the paired-end reads is d. This is illustrated in Figure 2.

Additionally, we associate each position of these pseudoreads with a relatively low quality score q in order to ensure that real reads have higher priority during the genome assembly process.

By default, whenever the opposite is not stated explicitly, we set $m = 100$, $n = 30$, $d = 1000$, and $q = 10$. The quality score is on the Phred scale from 0 to 93. We store the pseudoreads together with the quality scores as FastQ files [19] so that they can be used as input for the genome assembler Velvet.

2.2. Assisted Assembly. The second step of our approach leads to generation of assisted assembly contigs. To this aim, we combine pseudoreads generated in the previous step and experimental reads in one data set. Next, this data set is used as an input for a genome assembler. In principle, any assembler can be applied, but we use Velvet with its default parameters. However, the user may set values of the parameters according to his or her needs.

2.3. Editing the Reference. While editing the reference based on the alignment of the contigs produced by MUMmer, we have to take into account that contigs may be mapped ambiguously to the reference; that is, the same contig may be mapped to several segments of the reference. Moreover, the regions covered by different contigs may overlap and therefore some segments of the genome may be covered by several contigs. We resolve this ambiguity in two steps.

First, for each contig, we search for its *best* mapping to the reference. Conceptually, we can measure the quality of a mapping by the number of identical bases between the contig and the corresponding segment of the reference. This is estimated as

$$Q_{\text{map}} = L \times \text{idy}^{(\text{ref})}, \tag{1}$$

where L denotes the length of the mapped segment of the contig and $\text{idy}^{(\text{ref})}$ is the percentile identity between the contig and the corresponding reference segment as outputted by MUMmer. For each contig, out of its several mappings, we select the one that has the highest Q_{map} score.

Even though there is no theoretical guarantee that a particular contig corresponds to that segment of the genome to which it was mapped with highest Q_{map} score, we argue that, on one hand, the higher the identity is, the higher the likelihood that the mapping is correct is (i.e., the contig really originates from that segment of the genome to which it is mapped); on the other hand, the longer the mapped subsequence of the contig is, the higher the likelihood that the mapping is correct is. Therefore, the higher the above quality score is, the higher the likelihood of correct mapping is. Thus, we select for each contig the segment of the genome that has the highest Q_{map} score.

As one can see in Figures 5 and 7, the ratio of ambiguously mapped contigs varies between 5% and 12% in most of our experiments. An exception is the case of *A. thaliana*, for which the proportion of ambiguously mapped contigs is between 30% and 40%. After selecting the best mapping for each contig, the remaining ambiguity may only arise from overlapping contigs as illustrated in Figure 3. According to our observations, selecting the best mapping for each contig greatly reduces the number of those genomic positions that are covered by multiple contigs. In particular, in both cases of *A. thaliana* and *D. melanogaster*, the selection of the best mapping of each contig reduced the number of multiply covered genomic positions by ≈90%. Furthermore, the overlapping segments of two contigs typically contain exactly the same or very similar genomic sequences. Therefore, the selection of the best mapping for each contig is able to eliminate vast majority of the ambiguity. In the light of these observations, Figure 3 shows an exceptional situation, in which two contigs overlap and the overlapping segments correspond to notably different genomic sequences. Despite the fact that such situations are exceptionally rare, in order to produce the edited reference, such ambiguity must be resolved. One possibility to resolve such ambiguity is to use the aforementioned quality scores and to prioritize the contig with higher Q_{map} score. In our prototypical implementation of the pipeline, we used an even simpler method: we resolved the ambiguity remaining after the selection of the best mapping in a greedy fashion by preferring the beginning of the contigs to the ends of the contigs as illustrated in Figure 3.

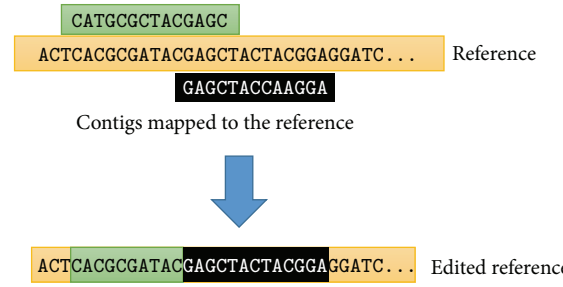


Figure 3: Resolution of ambiguity. First, for each contig, its best mapping is determined, and then the remaining ambiguity is resolved in greedy fashion by giving priority to the beginning of the contigs as shown in the figure.

After resolving the ambiguity, the edited reference is produced by replacing the segments of the reference by the mapped contigs (or their segments).

2.4. Software. We implemented RECORD using Perl and Java programming languages. The main program is implemented in Perl programming language. The main program calls Velvet and the modules, for generation of pseudoreads and reference editing.

3. Results and Discussion

Our approach does not aim to reproduce the reference genome (which is used as input anyway), but we aim to recover the true genome of the organism which is unknown in case of real experiments. Consequently, the evaluation of any assembly software is inherently difficult. Therefore, in the following sections, we present evaluation on both simulated and real data. In case of simulated data, a gold standard is

available, while the experiments on real data will show that our approach may be useful in real applications.

Next, we present the results of the experimental evaluation of our approach.

3.1. Baselines. In the experiments presented in the subsequent sections, we used two genome assemblers, Velvet [17] and Amos [9], as baselines. Velvet is a de novo genome assembler; that is, it assembles the genome directly from the experimental reads, whereas Amos is one of the most popular assisted genome assembly software tools; that is, Amos uses both the experimental reads and the reference genome of a genetically related organism in order to reconstruct the genome of the studied organism. Throughout the description of the experiments, with Velvet we refer to the case of using Velvet as standalone application, even though our approach, referred to as RECORD, uses by default Velvet as a component of the proposed pipeline.

We also tried to use further assisted genome assemblers, such as ARACHNE [11, 12] and IMR/DENOM [13]. While these softwares may excel in various general settings (such as using the reference genome of a species to reconstruct the genome of an other species), as far as we can judge, they do not seem to fit to our special setting of relatively *low coverage* (i.e., few experimental reads) and *very* closely related genomes. For example, in some cases, the outputted genome was the reference genome, which, on one hand, may be considered as reasonable if the actual genome and the reference genome are *highly* similar (i.e., they are *almost* the same); on the other hand, this is a trivial solution for the assisted assembly problem as the reference is one of the inputs of reference-assisted assembly methods.

3.2. Evaluation on Simulated Data. We simulate the scenario that the reference genome is given and we aim to reconstruct the actual genome of the studied organism, which we call *target genome*. In particular, we used the Evolver software tool [20] to generate the target genome. We used the genome from the example that comes with Evolver. This is an artificial mammalian genome of size of 30 megabases (Mb). The genome has three chromosomes. In order to allow for an unbiased evaluation, we produced the evolved genome following the example attached with Evolver. We used the original genome, that is, ancestral genome, as the reference genome, and we considered the evolved genome as the target genome. We generated one million paired-end short reads of length of 70 with wgsim [21] from the target genome. Subsequently, we tried to reconstruct the target genome from the generated paired-end reads and the reference genome both with our approach and two other state-of-the-art genome assemblers. Throughout the experiments on simulated data, we used Velvet with k-Mer size of $k = 21$. Finally, we compared the outputs of the assemblers with the target genome and quantitatively measured the quality of each of the assemblers according to the following criteria:

(1) TL, the total length of the assembly in Mb.

(2) N50; that is, we consider the set of largest contigs that together cover at least 50% of the assembly, and out of these contigs the length of the shortest one is denoted as N50.

(3) Error = 100% − IDY, where IDY is the percentile identity between the target genome and the genome reconstructed by the assembler. (Please note that IDY is different from $\text{idy}^{(\text{ref})}$. While $\text{idy}^{(\text{ref})}$ denotes the identity between an assembly contig and the corresponding segment of the *reference* genome, we use IDY to denote the identity between the output of the assembly and the *target* genome.) In order to calculate IDY, we map the genome reconstructed by the assembler to the target genome using the MUMmer software tool [18], and we calculated the weighted average of the percentile identities between the mapped segments and the target genome as outputted by MUMmer. In the weighted average, we used the length of the mapped segments as weights.

(4) Number of identical bases, which we calculated as IDY × TL.

Table 1: Evaluation on simulated data.

Assembly	TL (Mb)	N50	Error (in %)	Id. Bases (Mb)
Contigs				
Velvet	18.20	213 b	0.85	18.05
Amos	28.82	1834 b	2.09	28.22
RECORD	25.81	2055 b	0.41	25.70
Edited reference				
Velvet	30.00	10 Mb	1.39	29.58
Amos	30.00	10 Mb	1.03	29.69
RECORD	30.00	10 Mb	0.59	29.82

Both in case of our approach and in case of the baselines, we evaluated both the contigs and the edited reference resulting from using the contigs. In case of evaluating edited reference for the baselines, we simply used the contigs outputted by the baselines in the third step of our approach and produced the edited reference.

Table 1 summarizes our results. The columns of the table show the total length (TL) of the assembly, N50, error, and the number of identical bases. As one can see, our approach, RECORD, is competitive with the other assemblers: considering the contigs produced by our pipeline, they have the highest N50 and the lowest error rates, while the edited reference produced by RECORD has the overall highest number of identical bases with the target genome.

In a subsequent experiment, we varied the number of reads used for the assembly and evaluated the resulting contigs. These results are shown in Figure 4. The diagram (a) shows the number of bases in the target genome that are covered by the assembly contigs as function of the number of reads that were used. It is important to note that while Amos can provide overall better coverage of the sequence, it requires more reads (>500 k) for that. In the lower range of the number of reads available, it is outperformed by RECORD. It may be

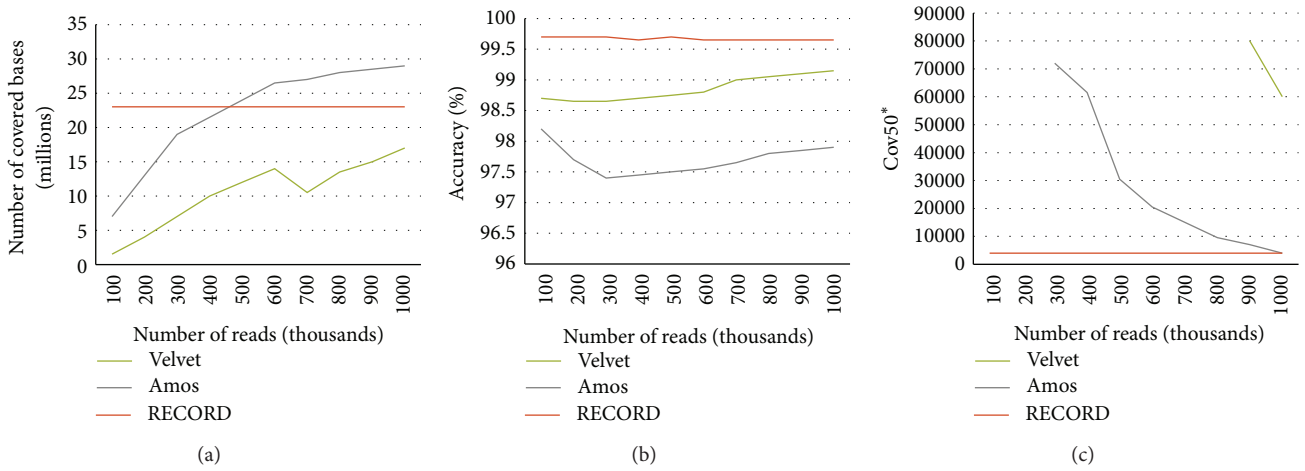


Figure 4: Comparison of the proposed approach (RECORD) with two state-of-the-art genome assemblers on data simulated with wgsim. In this experiment, we consider the evolved genome produced by Evolver as the target genome; the reference genome is the ancestral genome. The diagrams show the performance of the examined approaches according to various criteria as the function of the number of simulated reads that were used for the assembly. The diagram (a) shows the number of covered bases of the target genome; the diagram (b) shows the accuracy, that is, overall percentile identity between the assembly contigs and the corresponding segments of the target genome, while the diagram (c) shows the number of those largest contigs that together cover at least 50% of the target genome.

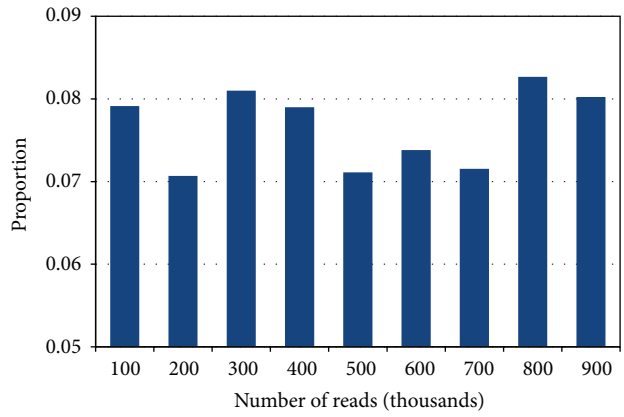


Figure 5: Proportion of ambiguously mapped contigs (before the selection of the best mapping for each contig) in case of various numbers of simulated reads.

relevant for practical applications as the cost of the experiment usually depends on the number of reads produced. While in this simulated case the number of reads is relatively low for today NGS technology standards, it might be still relevant in multiplexing scenarios where multiple experimental DNA libraries are barcoded and pooled in a single sequencing lane.

The second diagram (b) shows the overall percentile identity between the target genome and the contigs of the assembly. The third diagram (c) shows the number of those largest contigs that together cover at least 50% of the target genome. As one can see, if only relatively few reads are available, our approach, RECORD, systematically outperforms the baselines by producing larger contigs, the most accurate and most complete assembly.

In order to analyze our approach in more detail, we show in Figure 5 the proportion of ambiguously mapped contigs (before the selection of the best mapping for each contig). As one can see, the proportion of ambiguously mapped contigs varies between 7% and 8.5%.

We note that, from the point of view of applications, there is a substantial difference between the execution times of RECORD and Amos. For example, when using 300 thousand reads, producing the edited reference took approximately 1 hour for our approach, whereas it took 16 hours for Amos. We emphasize that this observation refers to the practical application of the software but not to the overall (theoretical) computational costs: much of the observed difference may be attributed to the fact that Velvet, which is used by default as assembler in the proposed pipeline, is able to run in parallel on multiple cores, whereas Amos can be used on one core at a time. Due to the fact that RECORD uses a de novo assembler as a component of the proposed pipeline, our approach is limited to middle-sized genomes that are closely related to the reference genome; therefore it is currently not applicable to the human and comparable genomes.

3.3. Evaluation on Real Data. The primary goal of the evaluation on real data was to show that our approach can be applied in real experiments.

3.3.1. Assessment of the Accuracy in Comparison to the Baseline. As mentioned previously, in real-world settings, there is usually no gold standard available. Therefore, the assessment of the accuracy of the genome produced by any assembler is inherently difficult. For this reason, in the subsequent experiment, we evaluate the accuracy of the proposed method on real data indirectly. In particular, we assess the quality of the contigs and, more importantly, we compare our approach to the baselines in the following setting: we examine how well

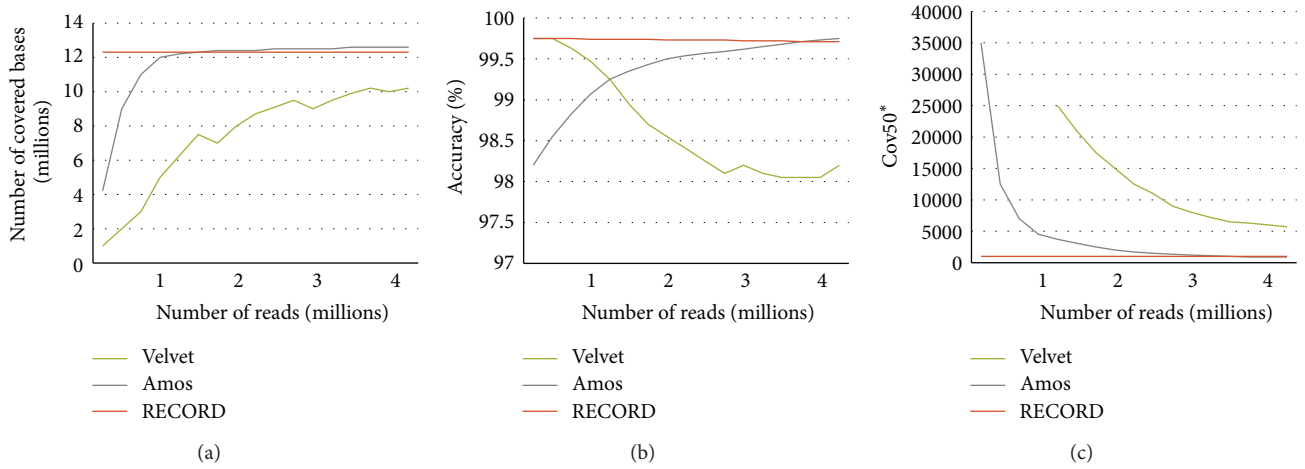


Figure 6: Comparison of the proposed approach (RECORD) with two state-of-the-art genome assemblers on real data. In this experiment, we compared assemblies resulting from various number of experimental reads to the assembly which is produced by Amos using all the experimental reads; that is, the target genome is the assembly produced by Amos using all the reads. In this case, the reference genome exhibits 99.7 percent identity with the result of Amos which is used as the gold standard. The diagrams follow the same structure as the one in Figure 4.

we can reconstruct the genome using relatively small subsets of all the available reads. These subsets are uniform random samples taken from the set of all the reads: each read has the same probability of being included in the sample. Paired-end reads are sampled together with their mates; that is, either both sequences corresponding to a particular paired-end read are selected or none of the sequences of that paired-end is selected.

In the aforementioned context, as gold standard, that is, target genome, we consider the genome produced by Amos when using *all* the reads for the assembly. We note that this leads to an evaluation in which Amos has an inherent advantage against our approach, as unfortunately we cannot have an unbiased reference.

We used real-world experimental reads graciously provided by dr Andrzej Dziembowski's group, coming from an unpublished ChIP-seq experiment in a yeast species. The data contained approximately 4.5 million paired-end short reads of length of 100.

Figure 6 shows the results. The diagrams follow the same structures as the ones presented at the end of Section 3.2; that is, the diagram (a) shows the number of bases in the target genome that are covered by the assembly contigs as function of the number of reads that were used. The second diagram (b) shows the accuracy, that is, the overall percentile identity between the target genome and the contigs of the assembly. The third diagram (c) shows the number of those largest contigs that together cover at least 50% of the target genome. In all the three diagrams, the horizontal axis shows the size of the sample (i.e., the number of paired-end reads) used to assemble the genome. As one can see, our approach, RECORD, systematically outperforms the baselines in terms of accuracy and coverage of the genome. Note that, in case of using very few reads, Velvet achieves as good accuracy as our approach; however, the contigs it produces have very low coverage.

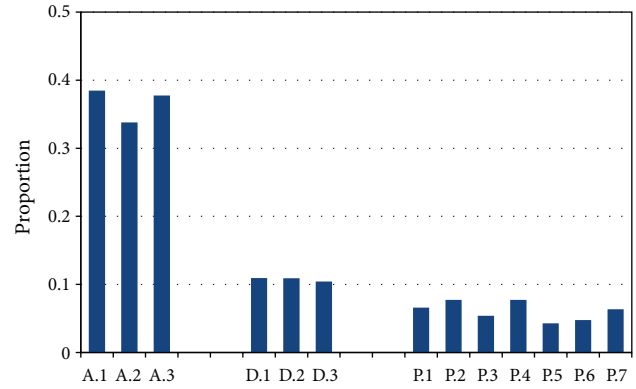


Figure 7: Proportion of ambiguously mapped contigs (before the selection of the best mapping for each contig) in case of experiments on publicly available data sets.

3.3.2. Characteristics of the Assembly of Publicly Available Data Sets. In order to assist reproducibility of our results, we used publicly available real short read data from the NCBI Short Read Archive. We used data originating from three different species: plant (*A. thaliana*), fly (*D. melanogaster*), and yeast (*S. pombe*). The identifiers of the short read collections are shown in the third column of Tables 2 and 3.

We set the k-mer size for the assembly, that is, the second step of our approach, in accordance with length of the short reads in the archive and the (approximate) size of the target genome. In particular, similarly to the previous experiments, we set $k = 21$ for yeast (short read length = 44), while we used slightly larger settings for the other two species: we set $k = 45$ in case of flower (short read length = 80) and $k = 25$ for fly (short read length = 36).

Tables 2 and 3 show the most important characteristics of the resulting assembly contigs and the edited reference.

Table 2: Assembly of real experimental reads (contigs).

Species	Number	Experimental reads	Length (Mb)	# ctgs	N50	Cov
A. thaliana	A.1	[SRR402840, SRR402839]	113.2	250148	19464	39.6x
	A.2	[SRR402842, SRR402841]	112.4	296438	18916	24.4x
	A.3	[SRR402844, SRR402843]	113.1	243375	20494	30.5x
D. melanogaster	D.1	[SRR066834, SRR066831]	122.6	460842	58648	2.6x
	D.2	[SRR066835, SRR066832]	122.6	461044	58535	2.1x
	D.3	[SRR066836, SRR066833]	122.6	460200	58415	2.4x
S. pombe	P.1	[SRR948260, SRR948250]	15.2	10384	8934	32.3x
	P.2	[SRR948261, SRR948251]	12.2	8129	11132	18.3x
	P.3	[SRR948262, SRR948252]	12.1	8571	6696	26.5x
	P.4	[SRR948266, SRR948272]	12.1	7589	10144	21.5x
	P.5	[SRR948267, SRR948273]	12.0	9214	4927	31.6x
	P.6	[SRR948268, SRR948274]	12.0	8964	5696	28.1x
	P.7	[SRR948269, SRR948275]	12.1	8269	8918	27.4x

Table 3: Assembly of real experimental reads (edited reference).

Species	Number	ed.len. (Mb)	% ref	% asm	# ctgs	% IDY
A. thaliana	A.1	109.9	91.8	97.3	51769	99.967
	A.2	106.4	89.0	95.5	58052	99.918
	A.3	109.8	91.8	97.3	54185	99.972
D. melanogaster	D.1	117.4	82.1	95.8	50598	99.985
	D.2	117.0	81.8	95.4	50606	99.986
	D.3	117.2	82.0	95.6	50630	99.986
S. pombe	P.1	12.0	95.1	78.9	3548	99.995
	P.2	12.0	95.2	98.4	2931	99.994
	P.3	12.0	95.0	99.2	4249	99.996
	P.4	12.0	95.3	99.2	3054	99.994
	P.5	11.9	94.6	99.2	5287	99.996
	P.6	12.0	94.9	100.0	4762	99.996
	P.7	12.0	95.3	99.2	3465	99.994

In particular, the fourth column of Table 2 shows the total length of the assembly contigs; the fifth column shows the number of all the contigs, while in the sixth column the N50 of the contigs is shown. The last column shows the coverage of experimental reads calculated as follows:

$$\text{Cov} = \frac{\text{read length} \times \text{number of reads}}{\text{genome size}}. \quad (2)$$

The third column of Table 3 shows the total length of the replaced segments, while the fourth and fifth columns show in percent the ratio of the length of the replaced segments relative to the length of the reference and the total length of the assembly contigs, denoted as % ref and % asm, respectively. The sixth column of Table 3 shows the number of contigs that were used while editing the reference. The last column of Table 3 shows the overall percentile identity between the edited reference and the original reference.

Figure 7 shows the proportion of ambiguously mapped contigs (before the selection of the best mapping for each contig) for each experiment shown in this section. As one can see, the proportion of ambiguously mapped contigs varies between 4% and 8% in case of *S. pombe*; it is around 10% in case of *D. melanogaster*; and it is remarkably higher, around 35%, for *A. thaliana*.

The results in Table 3 show that the total length of the assembly is close to the genome size, indicating the completeness of the assembly. However, the relatively large number of contigs in the raw assembly output can be seen as an indication that the assembler had difficulties with particular regions of the genome, and therefore a large number of short fragments may have been produced. This is especially visible in the case of *D. melanogaster*, where two factors influencing the quality of assembly are combined: low read coverage and low read length.

According to the proposed procedure of editing the reference, a contig may be left out if it can not be mapped to the reference or if MUMmer considers it too short to produce a useful alignment. As we can see, the number of contigs contributing to the edited reference is substantially less than the total number of contigs. However, in terms of length, almost the entire assembly is used; for example, in each of the *D. melanogaster* data sets the edited assembly utilizes ~11% of all contigs, covering over 95% of the assisted assembly. This shows that reference editing relies on a moderate amount of long contigs rather than on a bulk of short ones.

The edited part of the genome in *A. thaliana* is slightly smaller than in *D. melanogaster*, but the number of contributing contigs is slightly larger. Therefore, contigs obtained for the former organism are generally shorter than those obtained for the latter (this is also in accordance with the observation that N50 of *A. thaliana* is ~3× lower than N50 of *D. melanogaster*). Shorter contigs are more likely to be nonuniquely mapped, as we can observe on Figure 7; the proportion of ambiguously mapped contigs is similar to the proportion obtained in simulated data for *S. pombe* and *D. melanogaster*, while it is remarkably higher for *A. thaliana*.

Percentages of replaced segments in genome editing (% ref and % asm) are also similar to those observed in simulated data for two of our species (*A. thaliana* and *S. pombe*), while

they are slightly lower for *D. melanogaster*. This behavior is explained by the difference in the coverage, which is in *D. melanogaster* an order of magnitude lower than in the two other species. The results indicate that the outputted genomes are closely related to the reference. This is expected, since the genomes of individuals are close to a reference genome of the respective species.

Overall, the results on real-world data are similar to those on simulated data (in some respects, e.g., N50, even better). Visibly more variability is observed between results on real data sets with different characteristics: read length, coverage, and so forth.

4. Conclusions

In this paper, we proposed a new approach for reference-assisted assembly of closely related genomes. Our approach takes into account that the actual genome of the studied organism may be slightly different from the reference genome of that species leading to potentially fewer errors in downstream analyses of the sequenced read abundances.

We have assessed the performance of our method on an artificially simulated mutated eukaryotic genome, showing that RECORD produces contigs with very low error rate (less than 0.5 percent) and after merging them with the original assembly leading to error rates smaller than in simpler de novo assembly technique (Velvet) as well as more general assisted assembly approach (Amos).

Further examination of the results in comparison to Amos and simple Velvet indicated that our approach is most useful in the case where we have relatively few reads at our disposal; both of the competing tools struggled with the data sets where the number of reads was low.

The same seems to be true in case of a real data set that we analyzed in Section 3.3.1. Even though the numbers of reads are much higher, we can still see the difference between our method and the more traditional approaches. Even though the genome size is small, we can see that RECORD shows clearly superior accuracy with up to 3 million reads and all measures are clearly better at approximately 1 million reads.

Finally, we apply RECORD to more than 10 publicly available data sets from Short NCBI Read Archive to show its applicability in practical situations. We can see in all cases that not only is RECORD able to produce results for much larger genomes (up to 140 Mb) but the estimated divergence between the examined genome and the reference is close to one percent where we can expect RECORD to perform better than its examined alternatives.

We provide a prototype implementation of this approach as a set of scripts. It is available for download at our supplementary website together with most of the data published in the study allowing the readers to replicate our results and adapt the method for specific applications.

Availability and Requirements

Project name: RECORD Genome Assembler

Project home page: http://sourceforge.net/projects/record-genome-assembler/

Operating system(s): Linux

Programming language: Perl, Java

Other requirements: Velvet, MUMmer

License: Open Source

Any restrictions to use by nonacademics: no.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Authors' Contribution

The idea of using assisted assembly in the context presented in the paper originates from Norbert Dojer. Krisztian Buza implemented RECORD, performed the experiments, and wrote the first draft of the paper. Bartek Wilczyński supervised the implementation of RECORD and the experiments. All the three authors contributed to the discussions around the paper, proofread the paper, and contributed to the final text of the paper.

Acknowledgments

This work was partially supported by the research (Grant no. ERA-NET-NEURON/10/2013) from the Polish National Centre for Research and Development and by the Polish Ministry of Science and Education (Grant no. [N N519 6562 740]). Krisztian Buza acknowledges the Warsaw Center of Mathematics and Computer Science (WCMCS) for funding his position. The authors are thankful to Andrzej Dziembowski and Aleksandra Siwaszek from Institute of Biochemistry and Biophysics, Polish Academy of Sciences, for providing them with some of their unpublished ChIP-Seq input data from *S. pombe* which they used in their evaluation.

References

[1] M. L. Metzker, "Sequencing technologies the next generation," *Nature Reviews Genetics*, vol. 11, no. 1, pp. 31–46, 2010.

[2] O. Morozova and M. A. Marra, "Applications of next-generation sequencing technologies in functional genomics," *Genomics*, vol. 92, no. 5, pp. 255–264, 2008.

[3] K. Cui and K. Zhao, "Genome-wide approaches to determining nucleosome occupancy in metazoans using MNase-Seq," in *Chromatin Remodeling*, vol. 833 of *Methods in Molecular Biology*, pp. 413–419, Springer, 2012.

[4] L. Song and G. E. Crawford, "Dnase-seq: a high-resolution technique for mapping active gene regulatory elements across the genome from mammalian cells," *Cold Spring Harbor Protocols*, vol. 2010, no. 2, 2010.

[5] M. J. Fullwood, M. H. Liu, Y. F. Pan et al., "An oestrogen-receptor-α-bound human chromatin interactome," *Nature*, vol. 462, no. 7269, pp. 58–64, 2009.

[6] P. J. Park, "ChIP-seq: advantages and challenges of a maturing technology," *Nature Reviews Genetics*, vol. 10, no. 10, pp. 669–680, 2009.

[7] Z. Wang, M. Gerstein, and M. Snyder, "RNA-Seq: a revolutionary tool for transcriptomics," *Nature Reviews Genetics*, vol. 10, no. 1, pp. 57–63, 2009.

[8] Q. Peng and A. D. Smith, "Multiple sequence assembly from reads alignable to a common reference genome," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 8, no. 5, pp. 1283–1295, 2011.

[9] M. Pop, A. Phillippy, A. L. Delcher, and S. L. Salzberg, "Comparative genome assembly," *Briefings in Bioinformatics*, vol. 5, no. 3, pp. 237–248, 2004.

[10] J. Kim, D. M. Larkin, Q. Cai et al., "Reference-assisted chromosome assembly," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 110, no. 5, pp. 1785–1790, 2013.

[11] S. Batzoglou, D. B. Jaffe, K. Stanley et al., "Arachne: a whole-genome shotgun assembler," *Genome Research*, vol. 12, no. 1, pp. 177–189, 2002.

[12] D. B. Jaffe, J. Butler, S. Gnerre et al., "Whole-genome sequence assembly for mammalian genomes: Arachne 2," *Genome research*, vol. 13, no. 1, pp. 91–96, 2003.

[13] X. Gan, O. Stegle, J. Behr et al., "Multiple reference genomes and transcriptomes for *Arabidopsis thaliana*," *Nature*, vol. 477, no. 7365, pp. 419–423, 2011.

[14] M. Kolmogorov, B. Raney, B. Paten, and S. Pham, "Ragout—a reference-assisted assembly tool for bacterial genomes," *Bioinformatics*, vol. 30, no. 12, pp. i302–i309, 2014.

[15] E. Bao, T. Jiang, and T. Girke, "Aligngraph: algorithm for secondary de novo genome assembly guided by closely related references," *Bioinformatics*, vol. 30, no. 12, pp. i319–i328, 2014.

[16] S. Gnerre, E. S. Lander, K. Lindblad-Toh, and D. B. Jaffe, "Assisted assembly: how to improve a de novo genome assembly by using related species," *Genome Biology*, vol. 10, no. 8, article R88, 2009.

[17] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[18] S. Kurtz, A. Phillippy, A. L. Delcher et al., "Versatile and open software for comparing large genomes," *Genome Biology*, vol. 5, no. 2, article R12, 2004.

[19] P. J. A. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice, "The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants," *Nucleic Acids Research*, vol. 38, no. 6, pp. 1767–1771, 2010.

[20] R. C. Edgar, G. Asimenos, S. Batzoglou, and A. Sidow, "Evolver: A whole-genome sequence evolution simulator," http://www.drive5.com/evolver.

[21] H. Li, "WgSim: a small tool for simulating sequence reads from a reference genome," https://github.com/lh3/wgsim.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 761063, 12 pages
http://dx.doi.org/10.1155/2015/761063

Research Article

Accelerating Smith-Waterman Alignment for Protein Database Search Using Frequency Distance Filtration Scheme Based on CPU-GPU Collaborative System

Yu Liu,[1] Yang Hong,[1] Chun-Yuan Lin,[2] and Che-Lun Hung[3]

[1]*School of Electronic Information Engineering, Tianjin University, Tianjin 300072, China*
[2]*Department of Computer Science and Information Engineering, Chang Gung University, Taoyuan 33302, Taiwan*
[3]*Department of Computer Science and Communication Engineering, Providence University, Taichung 43301, Taiwan*

Correspondence should be addressed to Che-Lun Hung; clhung@pu.edu.tw

Received 18 March 2015; Revised 18 August 2015; Accepted 26 August 2015

Academic Editor: Hai Jiang

The Smith-Waterman (SW) algorithm has been widely utilized for searching biological sequence databases in bioinformatics. Recently, several works have adopted the graphic card with Graphic Processing Units (GPUs) and their associated CUDA model to enhance the performance of SW computations. However, these works mainly focused on the protein database search by using the intertask parallelization technique, and only using the GPU capability to do the SW computations one by one. Hence, in this paper, we will propose an efficient SW alignment method, called CUDA-SWfr, for the protein database search by using the intratask parallelization technique based on a CPU-GPU collaborative system. Before doing the SW computations on GPU, a procedure is applied on CPU by using the frequency distance filtration scheme (FDFS) to eliminate the unnecessary alignments. The experimental results indicate that CUDA-SWfr runs 9.6 times and 96 times faster than the CPU-based SW method without and with FDFS, respectively.

1. Introduction

In bioinformatics, the sequence alignment has become one of the most important issues. When the biologists get an unknown sequence, in general they would compare this unknown sequence (denoted as query sequence) with the known database of sequences (denoted as database sequences) to find the similarity scores and then identify the evolutionary relationships among them. Needleman and Wunsch [1] proposed a dynamic programming method (abbreviated to NW algorithm) to solve the global alignment problem between two sequences in 1970. The Smith-Waterman (abbreviated to SW) algorithm, which was proposed by Smith and Waterman [2] in 1981, is designed to find the optimal local alignment, and it is enhanced by Gotoh [3] in 1982. Although Hirschberg's algorithm [4] can be used for these algorithms above to reduce the memory space requirement, the computing time increases by a factor of two. When the lengths of two sequences are m and n, respectively, the time complexities of both NW and SW algorithms are $O(mn)$, respectively, and their space complexities by adapting Hirschberg's algorithm can be both reduced from $O(mn)$ to $O(m)$, where m is assumed to be larger than n. Though the NW and SW algorithms guarantee the maximal sensitivity for the alignment, the cost is still expensive, especially for the computation time.

Several fast heuristic methods such as FASTA [5] and BLAST [6, 7] have been devised to reduce the computation time at the expense of sensitivity. However, the exponential increase in the number of known sequences has increased the search time for querying against a database. Recently many core architectures, such as FPGA [8–10], Cell/Bes [11–13], and Graphics Processing Units (abbreviated to GPUs) [14, 15], have gradually become more popular in bioinformatics. These new architectures make it possible to enhance the performance of sequence alignments by using the parallel computing technologies. The use of GPUs has gradually

mainstreamed in the high-speed computing field. The potential advantage of GPU is its thousands of cores, with their total computing power exceeding the architecture with few CPUs. Also NVIDIA released the compute unified device architecture (abbreviated to CUDA) model to allow the programmers to use the commonly used programming languages, such as C/C++, to develop the applications. Thus many efforts have been made to accelerate the SW computations by CUDA on GPU.

In 2008, Manavski and Valle [16] presented the first SW algorithm by CUDA for the protein database search on GPU. The proposed algorithm was enhanced by the GSW algorithm proposed by Striemer and Akoglu [17] in 2009. Ligowski and Rudnicki [18] also presented another SW algorithm for the protein database search on GPU in 2009. Liu et al. proposed CUDASW++1.0 [19] and CUDASW++2.0 [20] for protein database search in 2009 and 2010, respectively. In CUDASW++1.0, they defined the intertask parallelization (abbreviated to ITE) and intratask parallelization (abbreviated to ITR) techniques for the relationships of task-thread and task-thread block on GPU, respectively. In general, the performance of the ITE technique is better than that of the ITR technique; however the ITE technique requires more memory space and then it is suitable for short sequences. Khajeh-Saeed et al. [21] proposed a CUDA-based SW algorithm (abbreviated to CUDA-SSCA#1) by using the ITR technique and multiple GPUs in 2010. In 2011, Hasan et al. [22] proposed a GPU-based SW algorithm (abbreviated to HKA algorithm) for the protein database search by using new sequence database organization and several optimizations to reduce the number of memory accesses. Hains et al. [23] developed new ITE and ITR kernels of SW algorithm for the protein database search in 2011. Sandes and de Melo proposed CUDAling1.0 [24] and CUDAling2.0 [25] for comparing two huge genomic sequences on GPU in 2010 and 2011, respectively. Liu et al. [26] presented CUDASW++3.0 in 2013 for the protein database search by coupling the CPU and GPU SIMD instructions and doing the CPU and GPU computations concurrently.

With the development of next-generation sequencing (NGS) techniques, the NGS machines can generate more than 1000 million nucleotide short reads (DNA, mRNA, and small-RNA) of lengths around 30~50 bps or more in a single run. For several NGS applications, such as metagenomics, these short reads will be assembled into possible contigs with lengths of tens to hundreds. These contigs are then used to match several well-known databases, such as NCBI-nt database, in order to classify or filter out these contigs. For coding DNA or mRNA short reads, they also can be translated to proteins for the proteomic research or be used for the analysis procedure of transcriptome. Now, several databases have more than hundreds of thousands protein (or nucleotide) sequences with lengths of tens to hundreds (or hundreds to tens of thousands). However, the previous works mentioned above are not suitable for the protein database search with a lot of database sequences. There are two reasons. One is that most of these works are suitable for short query and database sequences by using the ITE technique. The other is that these works all do the SW computations one by one (seen as the brute force search), and the computation time will be large for comparing a query sequence with a lot of database sequences under the limited hardware resources. Therefore, it is important to provide new concepts and procedures for the protein database search problem.

There is a possible method to compare a query sequence with a lot of database sequences, called CUDA-SWf proposed by Lee et al. [27] in 2013. CUDA-SWf used the frequency distance filtration scheme (abbreviated to FDFS) [28] in the run-time to filter out the unnecessary alignments, and then the computation time by CUDA-SWf is improved up to 41%. However, CUDA-SWf is also designed by using the ITE technique. The FDFS in CUDA-SWf is to calculate the frequency vectors of query and database sequences on GPU at first; then it calculates the frequency distance for each pair of query and database sequences, and after that the database sequences that need to be compared (denoted as selected database sequences) should be transferred from GPU to CPU. Finally, these selected database sequences are sorted according to their lengths and then retransferred from CPU to GPU in order to do the SW computations. Hence, the computation time by CUDA-SWf may be large for a lot of database sequences.

In this paper, we will propose an efficient SW alignment method, called CUDA-SWfr, for the protein database search by using the ITR technique based on a CPU-GPU collaborative system. In order to avoid the unnecessary alignments, FDFS is also applied to CUDA-SWfr to enhance the computation performance. For most of bioapplications, the used database can be predownloaded and preprocessed according to the application requirements. Therefore, the frequency vectors of database sequences can be precalculated and then stored in the database. Before doing the SW computations on GPU, FDFS is executed on CPU by calculating the frequency distances for each pair of frequency vectors from a query and database sequences. After that, the query and selected database sequences are transferred from CPU to GPU. The computation time of this procedure can be overlapped with that of SW computations. The experimental results indicate that CUDA-SWfr runs 9.6 times and 96 times faster than the CPU-based SW method without and with FDFS, respectively. These results indicated that CUDA-SWfr is suitable for the protein database search with a lot of database sequences.

The rest of this paper is organized as follows. Section 2 briefly describes the background knowledge of CUDA-SWfr, including the SW algorithm, the CUDA programming model, the frequency vector and frequency distance, and the related works of SW algorithm by CUDA on GPU. Section 3 then introduces CUDA-SWfr consisting of the implementations of FDFS on CPU and the SW computations on GPU. Section 4 gives the experimental results to evaluate CUDA-SWfr without and with FDFS.

2. Background Knowledge

2.1. The SW Algorithm. The SW algorithm is designed to identify the optimal local alignment between two sequences (query and database sequences). The SW computation needs a substitution matrix, such as a series of BLOSUM [28] or

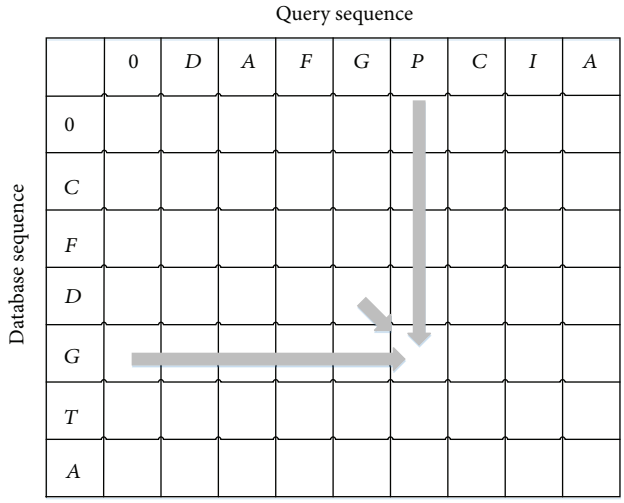


Figure 1: Dependency of calculating an alignment matrix $H(i, j)$.

PAM [29] matrices, and a gap-penalty function, such as the constant gap penalty or the affine gap penalty. The SW algorithm adopted in CUDA-SWfr is the same as that used in CUDA-SSCA#1 [21] with the affine gap penalty. Given two sequences S_1 and S_2 with lengths l_1 and l_2, respectively, the SW algorithm computes the similarity score in an alignment matrix $H(i, j)$ of these two sequences ending at positions i and j of sequences S_1 and S_2, respectively. The alignment matrix $H(i, j)$ is computed according to

$$H_{i,j} = \text{Max} \begin{cases} \text{Max}\left(H_{i-1,j-1} + S_{ij}, 0\right) \\ \underset{0<k<i}{\text{Max}}\left(H_{i-k,j} - \left(G_s + kG_e\right)\right) \\ \underset{0<k<j}{\text{Max}}\left(H_{i,j-k} - \left(G_s + kG_e\right)\right), \end{cases} \quad (1)$$

where $1 \le i \le l_1$, $1 \le j \le l_2$, and S_{ij} is the score in a substitution matrix, which is extracted according to a residue at position i in sequence S_1 and another residue at position j in sequence S_2. G_s is the gap opening penalty, G_e is the gap extension penalty, and k is the number of the extended gaps.

The maximum value of alignment matrix $H(i, j)$ indicates the similarity score between two sequences. The dependency of calculating an alignment matrix $H(i, j)$ is shown in Figure 1. As mentioned in the literature [21], formula (1) is the native concept of the SW algorithm. In order to improve the SW computation, the SW algorithm is modified as formula (2) according to the literature [3]. Formula (2) is more suitable for the parallel computing and the details of SW algorithm can be found in the literature [21]:

$$\begin{aligned} E_{i,j} &= \text{Max}\left(E_{i,j-1}, H_{i,j-1} - G_s\right) - G_e, \\ F_{i,j} &= \text{Max}\left(F_{i-1,j}, H_{i-1,j} - G_s\right) - G_e, \\ H_{i,j} &= \text{Max}\left(H_{i-1,j-1} + S_{ij}, E_{i,j}, F_{i,j}, 0\right). \end{aligned} \quad (2)$$

2.2. CUDA Programming Model. CUDA is an extension of commonly used programming languages, such as C/C++, in which users can write scalable multithreading programs for various applications. In general, the CUDA program is implemented in two parts: *Host* and *Device*. The *Host* part is executed by CPU, and the *Device* part is executed by GPU. The function executed on the *Device* part is called a *Kernel*. The *Kernel* can be invoked as a set of concurrently executing threads (abbreviated to TDs). These TDs are grouped into a hierarchical organization which can be combined into thread blocks (abbreviated to TBs) and grids (abbreviated to GDs). A GD is a set of independent TBs, and a TB contains many TDs. The size of GD is the number of TBs per GD, and the size of TB is the number of TDs per TB.

The TDs in a TB can communicate and synchronize with each other. TDs within a TB can communicate through a per-TB shared memory (abbreviated to sM), whereas TDs in the different TBs fail to communicate or synchronize directly. Besides sM, five memory types are per-TD private local memory (abbreviated to LM), global memory (abbreviated to GM) for data shared by all TBs, texture memory (abbreviated to TM), constant memory (abbreviated to CM), and registers (abbreviated to RG). Of these memory types, CM and TM can be regarded as fast read-only caches; the fastest memories are the register and sM. The GM, LM, TM, and CM are located on the GPU's memory. Besides sM accessed by a single TB and RG only accessed by a single TD, the other memory types can be used by all TDs. The caches of TM and CM are limited to 8 KB per streaming multiprocessor (abbreviated to SM). In the Kepler architecture, SM is also called SMX. The optimum access strategy for CM is all TDs reading the same memory address. The cache of TM is designed for TDs in order to improve the efficiency of memory access. The Fermi and Kepler architectures have real configurable L1 per SM and unified L2 caches among SMs. Hence, L2 caches can be accessed by GM and each SM can use the L1 caches and sM.

The basic processing unit in NVIDIA's GPU architecture is called the streaming processor (abbreviated to SP). In the Fermi and Kepler architectures, the basic processing unit is called CUDA cores. Many SPs perform the computations on GPU. Several SPs can be integrated into a SM according to various architectures, such as 32 and 192 SPs per SM for the Fermi and Kepler architectures, respectively. While the program runs the *Kernel*, the *Device* schedules TBs for the execution on the SM. The Single Instruction Multiple Thread (abbreviated to SIMT) scheme refers to TDs running on the SM in a small group of 32, called a warp (abbreviated to WP). The WP scheduler simultaneously schedules and dispatches instructions.

2.3. Frequency Vector and Frequency Distance. The frequency vector and frequency distance are proposed by Kahveci et al. [30] in 2004, and they are used to the whole genome alignment problem. In the literature [31], the frequency distance is also used to remove the sequences which are dissimilar between the query and subject sequences before performing the sequence alignment. In 2013, Lee et al. [27] used the frequency vector and frequency distance to remove the unnecessary SW alignments between the query and database sequences. Assuming that a query or database sequence s is composed of n kinds of nucleotides/amino

acids (denoted as the alphabet set), the frequency vector (abbreviated to FV) of s is defined as follows:

$$\mathrm{FV}(s) = f_s = [\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n], \quad (3)$$

where α_n represents the number of nth alphabets appearing in the sequence s. Assume that there are two DNA sequences u and v; the frequency distance (abbreviated to FD) of these two sequences is defined as follows:

$$\begin{aligned} \mathrm{FD}(f_u, f_v) &= |f_u - f_v| \\ &= (|A_u - A_v| + |T_u - T_v| + |G_u - G_v| + |C_u - C_v|). \end{aligned} \quad (4)$$

The FD is calculated from FVs of sequences u and v, which is the sum of differences for FVs of the alphabet set. For the biological sequences, the edit distance (abbreviated to ED) is a commonly used measurement to represent the difference between two sequences. In other words, a low edit distance means a high similarity score. The relation between FD and ED for sequences u and v is listed as follows:

$$\mathrm{FD}(f_u, f_v) \leq \mathrm{ED}(u, v). \quad (5)$$

In several bioapplications, a threshold as a lower bound of similarity score may be defined by the biologists to filter out the unwanted results. For example, in the homology modeling application, the similarity score should be larger than 35% between the template and target sequences. Moreover, in general, the sequence coverage ratio should be larger than 60% between the template and target sequences. Therefore, this threshold could be used as a factor to filter out unnecessary alignments between a query and database sequences. When we want to find the very similar sequences in a database for a query sequence, the value of ED is set to small. If a FD of a pair of query and database sequences is larger than ED, it means that this database sequence may be not similar to the query sequence. This database sequence can be omitted in the following SW computations. It is worth to note that a FD will be influenced by the sequence lengths between query and database sequences. When the length of a database sequence is longer than that of query sequence, a FD of these two sequences may be large due to their length difference. However, the possible subsequences in this database sequence may be similar to the query sequence. Hence, FDFS cannot be applied to the database sequence with a length longer than that of the query sequence in order to avoid the possible false negatives.

2.4. Related Works. Recently, many works have been proposed in the past to implement the SW algorithm on GPU. In the following, the brief descriptions of selected works have been made by considering the implementations and performance.

In 2006, Liu et al. [14] proposed the hardware implementation of the double affine Smith-Waterman (DASW) algorithm on a graphics card by using graphic API (OpenGL + GLSL). By only computing alignment scores, DASW achieved 24 M (millions or mega) DP cells per second on single NVIDIA GeForce 7800 GTX GPU card. Liu et al. [15] also presented an approach for the protein database scanning by using a graphics card (OpenGL + GLSL) to gain a high performance at the low cost. The proposed approach achieved more than 650 M cell updates per second (abbreviated to CUPS) on single NVIDIA GeForce 7800 GTX GPU card. Moreover, it ran 9 times and 15 times faster than SSEARCH [32] and OSEARCH [32], respectively. The above works are both proposed based on GPU by OpenGL (a GPGPU programming), not CUDA.

The SW-CUDA [16] precomputed a query profile stored in the TM to replace the query sequence and the substitution matrix. In SW-CUDA, each TD in a GD is used to do a SW computation by using the query profile and a database sequence. This process matched the definition of the ITE technique proposed by the literature [19]. Hence, SW-CUDA preordered the database sequences according to their lengths in order to balance the computing workload of each TD in a TB. The preordered database sequences are stored in the GM. For each SW computation, the alignment matrix is computed column by column and the similarity score is stored in the LM. The SW-CUDA only calculated the similarity score for a SW computation and achieved 1830 MCUPS (=1.8 giga-CUPS) and 3480 MCUPS (=3.48 GCUPS) on single and dual-NVIDIA GeForce 8800 GTX GPU cards, respectively, and it ran 2 to 30 times faster than any previous implementation of SW on GPU.

GSW algorithm [17] pointed out that the design of query profile is not suitable for GPU due to the limited size of TM on GPU. They still used the query sequence and the substitution matrix both stored in the CM to do the SW computations by using the ITE technique. They also proposed an efficient function with an ASCII code table to access the score in the substitution matrix. The database sequences are stored in the GM and the similarity scores are stored in the sM. The GSW algorithm ran 10 times faster than SSEARCH on single NVIDIA Tesla C870 GPU card; however, it is slower than Farrar's implementation [32]. Another SW algorithm [18] by using the ITE technique achieved 7.5 and 14.5 GCUPS on single and dual-NVIDIA GeForce 9800 GTX GPU cards, respectively.

In CUDASW++1.0 [19], the ITE technique means that each SW computation consisting of a pair of query and database sequences (denoted as a task) is assigned to one TD; the ITR technique means a task is assigned to one TB. Due to the limited memory on GPU, each TD can only process a pair of short sequences in a TB with many TDs by using the ITE technique. Conversely, a pair of long sequences can be processed by a TB with many TDs by using the ITR technique. However, the ITE technique can achieve better performance than the ITR technique due to more tasks concurrently executed on GPU. The alignment matrix is computed according to the diagonal direction in CUDASW++1.0. For the ITE technique, they also considered the effect of coalesced access in the GM. Therefore, a preordered database is also used in CUDASW++1.0 as the previous works above. However, these preordered database sequences should be rearranged and then stored in the GM. In CUDASW++1.0, a threshold is set to 3072 for the length of database sequence. If the length of database sequence

is less than the threshold, the SW computation is done by the ITE technique, otherwise by the ITR technique. CUDASW++1.0 achieved an average performance of 9.5 GCUPS and 14.5 GCUPS on single NVIDIA GeForce GTX 280 GPU card and dual-NVIDIA GeForce GTX 295 GPU card, respectively. However, the lengths of few sequences in the test Swiss-Prot protein database (release 56.6) are larger than the threshold [23]. CUDASW++2.0 [20] further optimized the performance of CUDASW++1.0 based on the SIMT abstraction of CUDA-enabled GPUs and achieved 17 GCUPS and 30 GCUPS on single NVIDIA GeForce GTX 280 GPU card and dual-NVIDIA GeForce GTX 295 GPU card, respectively. CUDASW++3.0 [26] achieved the maximum performance of 119.0 and 185.6 GCUPS on single NVIDIA GeForce GTX 680 GPU card and dual-GPU GeForce GTX 690 GPU card, respectively.

The alignment matrix is computed according to the row direction in CUDA-SSCA#1 [21]. The SSCA#1 benchmark was used to evaluate 5 kernels that are various permutations of the SW algorithm, including saving the alignment results or not. In order to save the alignment result of a SW computation, CUDA-SSCA#1 needs to spend more memory space by using a traceback procedure. The HKA algorithm [22] processed the SW computations by using the ITE technique and achieved 1.13 times better than CUDASW++2.0 in terms of GCUPS. Hains et al. found that CUDASW++1.0 only achieved 1.5 GCUPS when comparing the same query and database sequences on single NVIDIA Tesla C1060 GPU card by using the ITR technique. Their ITR kernel [23] obtained better performance than that of CUDASW++1.0.

3. Methods

The implementation of CUDA-SWfr can be divided into two parts: FDFS executed on CPU and the SW computations executed on GPU. The flowchart of CUDA-SWfr is shown in Figure 2. The details of these two parts are described in the following sections, respectively.

3.1. FDFS Executed on CPU. In CUDA-SWfr, there is a procedure to do FDFS on CPU. There are four steps in this procedure: (1) construct the preorder database sequences with FVs, (2) calculate the FV of query sequence, (3) calculate the FDs for each pair of query and database sequences, and (4) collect the query and selected database sequences and then they are transferred from CPU to GPU. In the following, these four steps are described in detail, respectively.

Step 1 (construct the preorder database sequences with FVs). In the previous works in Section 2.4, the database sequences should be preordered according to their lengths due to the ITE technique. It is unnecessary for CUDA-SWfr by using the ITR technique. However, the following calculation of FD will be influenced by the sequence lengths between the query and database sequences as mentioned in Section 2.3. The database sequences are sorted and then stored in a database with their FVs in order to accelerate the computations of Step 3. The computation time of this step is omitted in the experimental test as the previous works in Section 2.4.

Step 2 (calculate the FV of query sequence). When the biologists want to use CUDA-SWfr for a query sequence, the FV of query sequence is calculated in the run-time by using formula (3). Moreover, the length of query sequence is also recorded in a variable. The FV and length of query sequence will be used in Step 3.

Step 3 (calculate the FDs for each pair of query and database sequences). As mentioned in Section 2.3, FDFS cannot be applied to the database sequence with a length longer than that of the query sequence in order to avoid the possible false negative. Hence, when a database sequence has the length longer than that of the query sequence, it needs to be compared in the second part of CUDA-SWfr and it is not necessary to calculate the FD with the query sequence. When the length of query sequence is short, this way will reduce a lot of time for Step 3. However, when a database sequence has the length shorter than or equal to the query sequences, the FD should be calculated by using formula (4) with the query sequence. When the calculated FD is larger than a threshold of ED, this database sequence is not selected according to formula (5). The threshold is set by the user. For the effect of FDFS, there are two factors. One is the length of query sequence and the other is the threshold of ED. When the length of query sequence is short, the lengths of most of database sequences may be longer than it, and the number of selected database sequences is large; in other words, the effect of FDFS is small. Similarly, when the threshold of ED is set to large, the FDs of most of database sequences may be smaller than this threshold, and then the effect of FDFS is small; see the experimental results in Section 4.

Step 4 (collect the query and selected database sequences and then they are transferred from CPU to GPU). In this step, the FVs of query and selected database sequences and FDs of a pair of query and selected database sequences are not needed to be transferred from CPU to GPU. By FDFS, the transmission data (especially for the selected database sequences) can be reduced, and then the transmission time is decreased according to two factors mentioned above. Therefore, there are two advantages by using FDFS. One is to filter out the unnecessary alignments and the other is to reduce the transmission time. Moreover, the time by Steps 3 and 4 can be overlapped with the time of SW computations executed on GPU. After this step, the second part of CUDA-SWfr is used to do the SW computations on GPU.

3.2. SW Computations Executed on GPU. After the first part of CUDA-SWfr, the selected database sequences are stored in the GM and the query sequence is stored in the CM. When the length of query sequence is larger than the size of the CM, it is stored in the GM. In the Fermi architecture, it may be fast to access the query sequence in the GM due to the L2 caches. In CUDA-SWfr, the SW computations are made by using the ITR technique, and it means that the database sequences do not need to be rearranged before being stored in the GM. For the SW algorithm, a substitution matrix is needed. Hence, a substitution matrix is also stored in the CM. In order to access the score in a substitution matrix efficiently, the function with

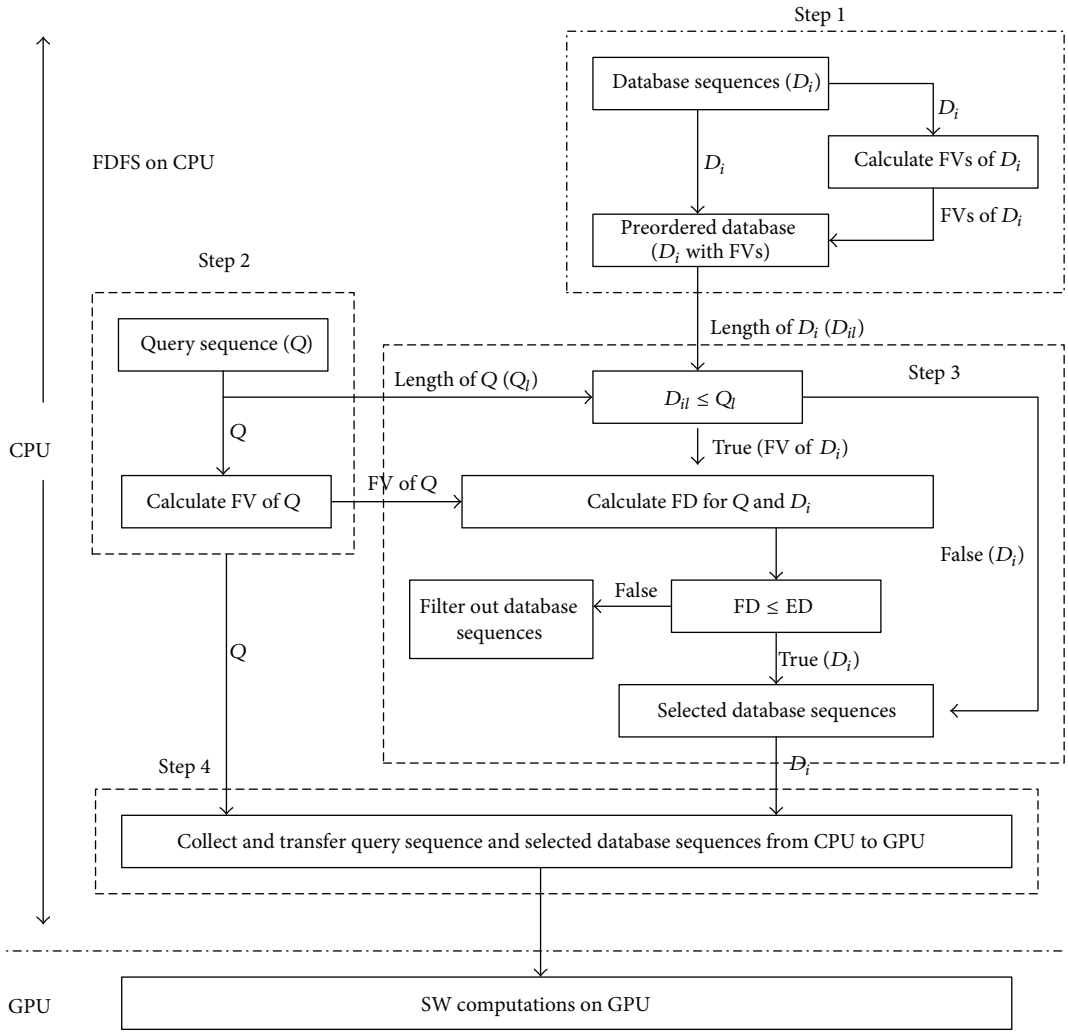


Figure 2: The flowchart of CUDA-SWfr.

an ASCII code table presented by Striemer and Akoglu [17] is also used in CUDA-SWfr.

By the ITR technique, all TDs in a TB are used to do a SW computation; it is a dynamic programming algorithm. According to the previous works and the SIMT scheme on GPUs, there are eight implementation types of dynamic programming by using the ITR technique on GPU [33]. These eight implementation types are synchronous row single thread (SRST), synchronous row multiple threads (SRMT), asynchronous row single thread (ARST), asynchronous row multiple threads (ARMT), synchronous diagonal single thread (SDST), synchronous diagonal multiple threads (SDMT), asynchronous diagonal single thread (ADST), and asynchronous diagonal multiple threads (ADMT). CUDA-SWfr adopts the SRMT type of dynamic programming to implement a SW computation by an assignment method, in which one row is assigned to a TB until all rows are assigned.

For a SW computation in a TB, according to formulas (1) and (2) and the dependency shown in Figure 1, this process can be divided into two stages. In the first stage, the value of each cell ($H(i, j)$) in a row is calculated according to the upper cell, upper-left cell and 0. All cells in a row are calculated by all of TDs in a TB. Assume that the length of a row is n and the number of TDs in a TB is t; the values of (n/t) cells are calculated by a TD, where n is larger than t in general. However, these values are only intermediate values without considering the left cell according to formula (1) and (2). These values in a row are stored in the sM. In the second stage, only a TD in a TB is used to correct the value of each cell in a row by considering its left cell. These two stages will be done repeatedly until all rows are assigned to this TB. The highest value among all cells in all rows is the similarity score for a query and a selected database sequence. For a query sequence, all of similarity scores or the best score will be copied from the sM to the GM, and then they could be transferred from GPU to CPU according to the user's requirement. In CUDA-SWfr, the number of TDs in a TB is set to 256 and the number of TBs in a GD is equal to the number of selected database sequences (or partial of them in order to overlap the computation time between FDFS and SW computations).

The goal of this paper is to accelerate SW computations for the protein database search by using FDFS based on a CPU-GPU collaborative system. As the previous works

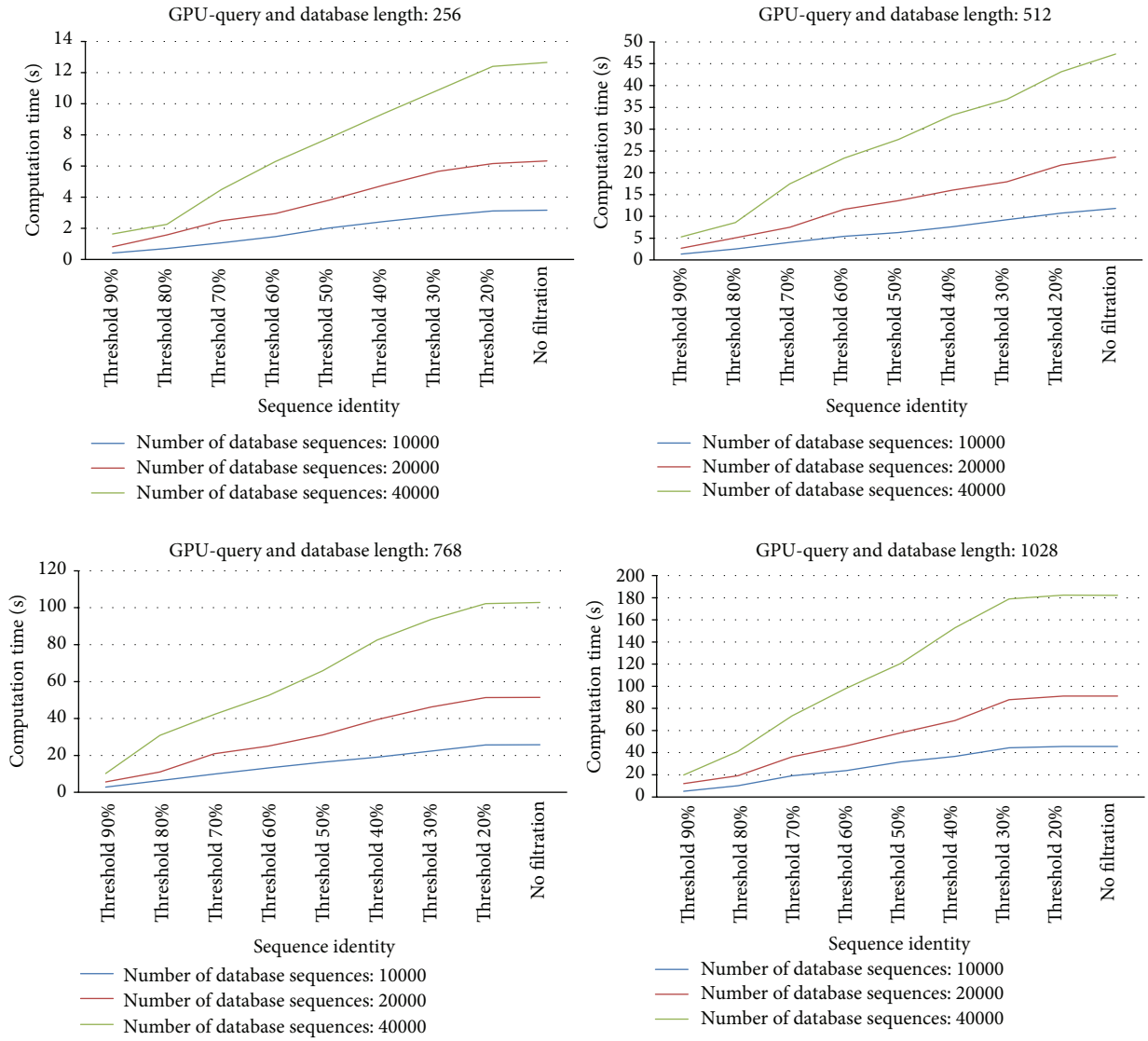


Figure 3: The overall computation time by CUDA-SWfr with various thresholds for 12 synthetic databases.

[16–19] in the past, the alignment quality is not the consideration in this work. Only similarity scores are calculated for each pair of query and database sequences by using the SW algorithm. The query and selected database sequences can be aligned again by using other alignment algorithms or tools with specific substitution matrices in order to obtain the good alignment results or other simulation results, such as the homology modeling.

4. Experiment Results

In this work, CUDA-SWfr is implemented by C+CUDA on single NVIDIA Tesla C2050 GPU card, with 448 SPs cores and 3 GB GDDR3 RAM. The *Host* device is Intel Xeon E5506 CPU of 2.13 GHz with 12 GB RAM running Ubuntu 10.04 operation system.

In order to evaluate the effect of FDFS in CUDA-SWfr, the synthetic sequences are generated in this work. A test query sequence with a length of 1028, NP_001116291, is a human protein sequence downloaded from the NCBI website (http://www.ncbi.nlm.nih.gov/). Three subsequences are extracted randomly from it with the lengths of 256, 512, and 768, respectively. These three subsequences and the original query sequence are used as the query sequences in the following tests. Based on these query sequences, 12 synthetic databases are generated and used in the experimental tests. These 12 synthetic databases can be classified into 4 groups according to the lengths: 256, 512, 768, and 1028. For each group, there are three synthetic databases with 10000, 20000, and 40000 database sequences. For each database sequence in a group, it is generated according to the corresponding query sequence with the same length. For each synthetic database, it has database sequences with 0%~100% mutations of its corresponding query sequence. For example, assume that a synthetic database has 40000 database sequences and the length is 1028. There are 4000 database sequences with 0%~10% mutations of query sequence, 4000 database sequences with 10%~20% mutations of query sequence, and

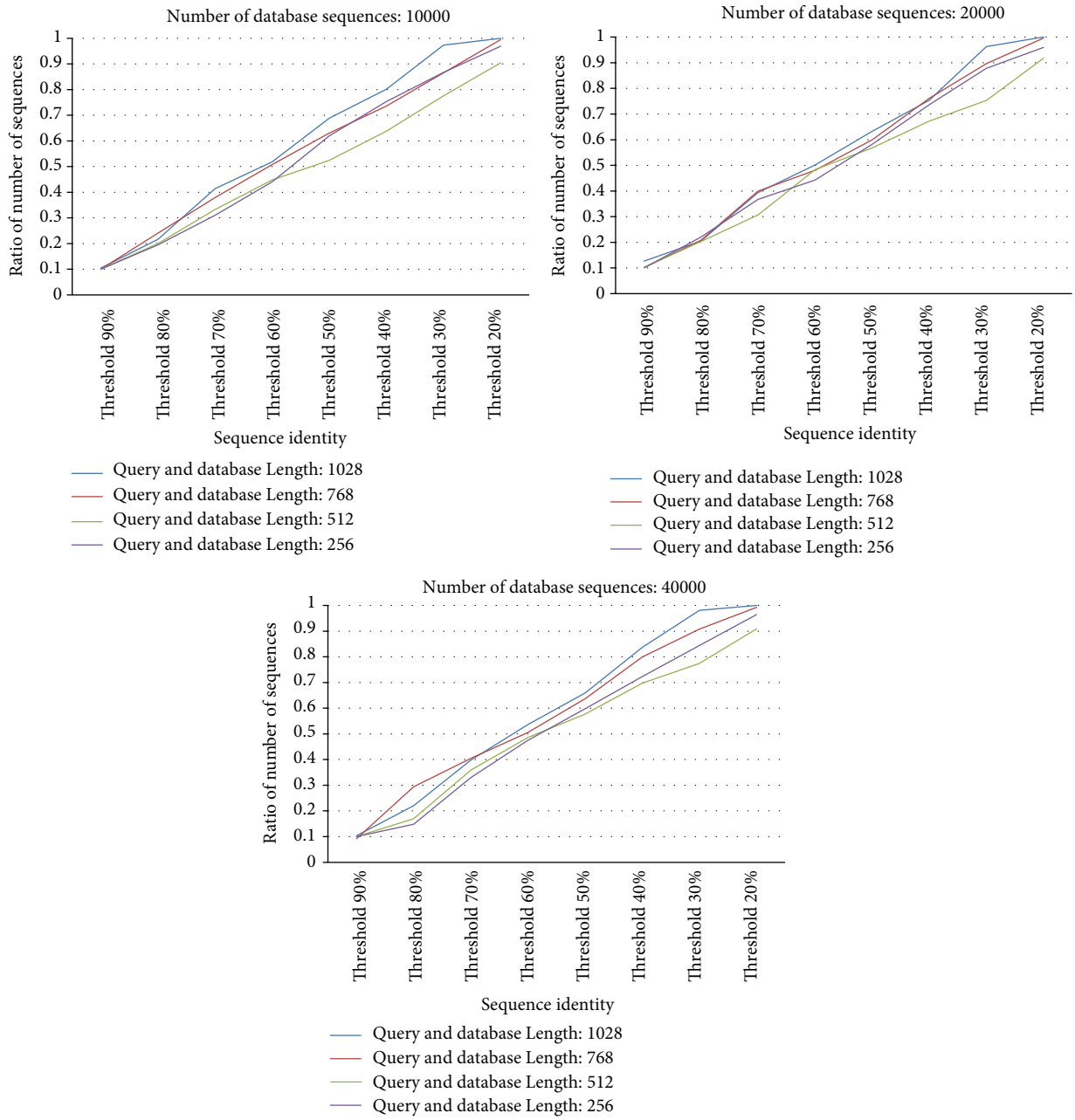


Figure 4: Ratio (%, number of selected database sequences/number of original database sequences) for 12 synthetic databases with various thresholds.

so on. When we want to generate one database sequence with 0%~10% mutations of query sequence, a number t is randomly selected between 0 (1028 × 0%) and 102 (1028 × 10%). If t is 57, it means that 57 positions in the query sequence are randomly selected at first, and then the residues in these 57 positions are changed to others by adding a constant (also randomly selected) with its original ASCII code. This work is done repeatedly until 40000 database sequences are generated for this synthetic database. In the following tests, the threshold of FDFS is the sequence identity. In other words, if a threshold is set to 90%, it means that the selected database sequence may be very similar to the query sequence.

Figure 3 shows the overall computation time (unit: second) by CUDA-SWfr with various thresholds for 12 synthetic databases. The threshold "None" means the SW computations by CUDA-SWfr without FDFS. From Figure 3, the computation time decreases when the threshold increases for 12 synthetic databases. When the threshold is set to 90%, the computation time by CUDA-SWfr is fastest for all of 12 synthetic databases. From Figure 3, it also shows that the computation time increases when the sequence length or the number of database sequences increases.

In order to observe the relationship between the threshold and the number of selected database sequences, Figure 4

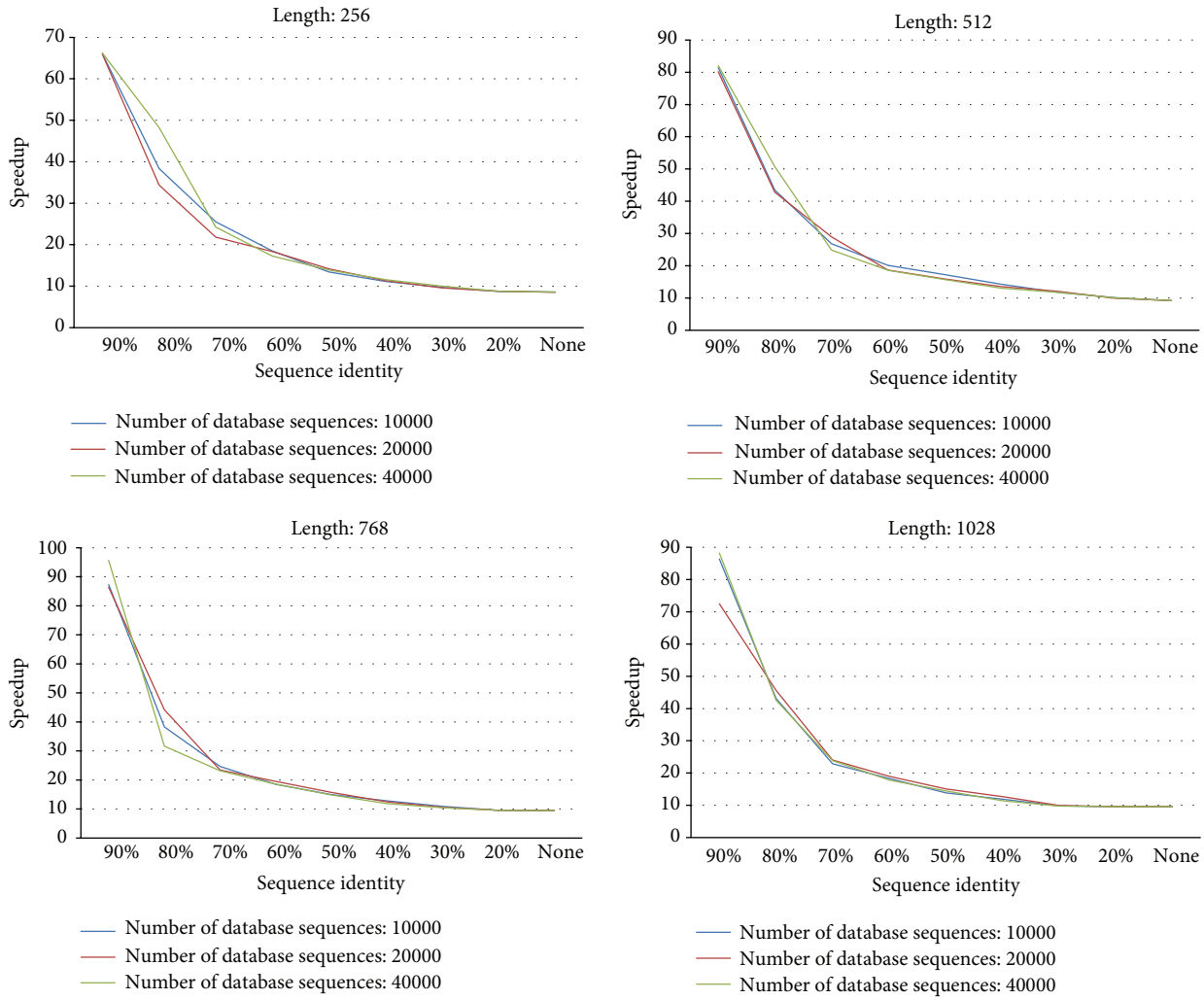


Figure 5: The overall speedup by comparing CUDA-SWfr with the CPU-based SW method under various thresholds for 12 synthetic databases.

shows the ratio (%) by dividing the number of selected database sequences by the number of original database sequences for 12 synthetic databases with various thresholds. From Figure 4, the ratio decreases when the threshold increases for 12 synthetic databases. When the threshold is 90%, in Figure 4, only 10% database sequences are selected for the SW computations on GPU for 12 synthetic databases. These results explain why the computation time by CUDA-SWfr is fastest when the threshold is set to 90%. From Figure 4, it also shows that the ratio is not influenced by the number of database sequences for 12 synthetic databases. Besides, when the length of query sequence is equal to that of database sequences, the ratio is also not influenced by the sequence length for 12 synthetic databases. However, the ratio will be influenced by the sequence length when the length of query sequence is not equal to that of database sequences. As mentioned in Section 2.3, when the length of database sequence is larger than that of query sequence, this database sequence should be compared with the query sequence. Hence, when database sequences have various lengths, for a query sequence, the ratio will increase when the number of database sequences, each having a length larger than that of query sequence, increases.

Figure 5 shows the overall speedups by comparing CUDA-SWfr with the CPU-based SW method under various thresholds for 12 synthetic databases. From Figure 5, the overall speedups by CUDA-SWfr without FDFS are about 8.5~9.6 times. The best speedup by CUDA-SWfr with FDFS achieved 96 times. From Figure 5, the speedup increases when the threshold increases for 12 synthetic databases. From Figure 5, it also shows that the speedup is not influenced by the number of database sequences for 12 synthetic databases. Besides, when the length of query sequence is equal to that of database sequences, the speedup is also not influenced by the sequence length for 12 synthetic databases. The observations in Figure 5 match those of Figures 3 and 4. Unlike the previous works that use the GCUPS to evaluate the performance, CUDA-SWfr only considered the computation time and the speedups. There are two reasons. First, the overall computation workload (DP cells)

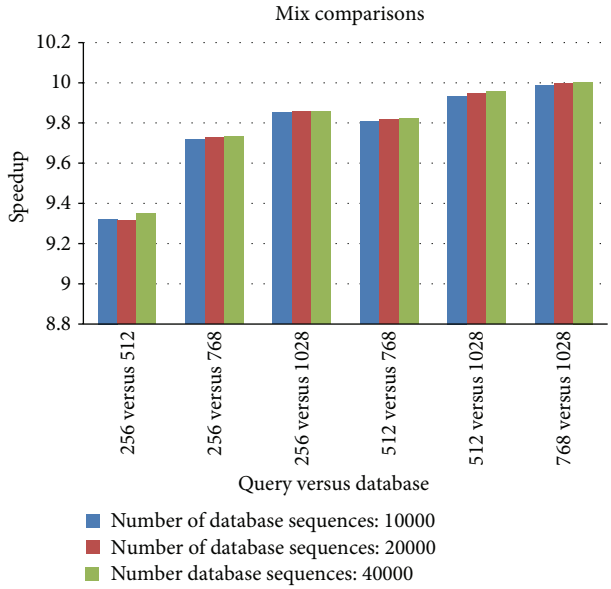


Figure 6: The overall speedup by comparing CUDA-SWfr with the CPU-based SW method for various combinations.

by CUDA-SWfr is difficult to be estimated due to filtering out the unnecessary alignments by FDFS. Second, the overall computation time by CUDA-SWfr consisted of FV and FD calculations (Steps 2 and 3 in Section 3.1), SW computations (Section 3.2), and transmission time (Step 4 in Section 3.1 and Section 3.2). Hence, it is hard to estimate the correct value of GCUPS. Assume that the overall computation time is spent to calculate overall cells. CUDA-SWfr achieved the peak performance of 2.133 GCUPS by using the ITR technique.

When the length of query sequence is less than the length of database sequences, Figure 6 shows the overall speedup by comparing CUDA-SWfr with the CPU-based SW method for various combinations. The query sequence of length 256 was used to compare with the synthetic database of lengths 512, 768, and 1028, respectively; the query sequence of length 512 was used to compare with the synthetic database of lengths 768 and 1028, respectively; the query sequence of length 768 was used to compare with the synthetic database of length 1028. The overall speedups by CUDA-SWfr in Figure 6 are about 9.3~10 times. Since the length of database sequence is larger than that of query sequence, all of database sequences should be compared with the query sequence. These results are similar to those (8.5~9.6 times) by CUDA-SWfr without FDFS as shown in Figure 3. From Figure 6, the speedup increases when the length of database sequences increases. The reason is that the overall computation workload increases when the length of database sequences increases.

In this work, we did not use the real biological sequences to evaluate the effect of FDFS. There are two reasons. The first one is the lengths of real biological sequences are various. In order to avoid the possible false negative, FDFS cannot be applied to the database sequence with a length longer than that of the query sequence. Hence, it is hard to choose the suitable set of real biological sequences and then to prove the relationships of ratio-threshold and ratio-speedup on GPU. The second one is the range of similarity scores among real biological sequences is large. In practice, it shows that FDFS scheme is necessary to filter out the unnecessary alignments. However, in the experimental tests, it is hard to choose the suitable set of real biological sequences to prove the relationships of threshold-speedup on GPU.

5. Conclusions

In this paper, we presented a GPU-based SW alignment method, CUDA-SWfr, with FDFS by using the ITR technique. The FDFS is executed by the CPU capability to filter out the unnecessary alignments and the SW computations are made by the GPU capability. From the experimental tests, CUDA-SWfr ran 9.6 times faster than the CPU-based SW method without FDFS; it ran 96 times faster than the CPU-based SW method with FDFS. The effect by FDFS is influenced greatly by the threshold. When the threshold as the sequence identity is large, the number of selected database sequences is small which means that the effect by FDFS is visible. In other words, the effect by FDFS can be omitted with a small threshold. Besides, the effect by FDFS also could be influenced by the number of database sequences; each has a length larger than that of query sequence. CUDA-SWfr is suitable for the protein database search with a lot of database sequences, and the proposed FDFS can be integrated into other research works, such as Huang et al. [34] and Feng et al. [35], in recent years.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Part of this work was supported by the Ministry of Science and Technology under Grants MOST104-2221-E-182-050, MOST 104-2221-E-182-051, MOST103-2221-E-126-013, and MOST103-2632-E-126-001-MY3. This work was also supported in part by the National Natural Science Foundation of China under Grant 61373102 and Yunan Academician Funding. The authors would like to thank the anonymous reviewers and experts who discussed this work with them.

References

[1] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, vol. 48, no. 3, pp. 443–453, 1970.

[2] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

[3] O. Gotoh, "An improved algorithm for matching biological sequences," *Journal of Molecular Biology*, vol. 162, no. 3, pp. 705–708, 1982.

[4] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communication of The ACM*, vol. 18, pp. 341–343, 1975.

[5] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 85, no. 8, pp. 2444–2448, 1988.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.

[7] S. F. Altschul, T. L. Madden, A. A. Schäffer et al., "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, no. 17, pp. 3389–3402, 1997.

[8] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. L. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," *Bioinformatics*, vol. 21, no. 16, pp. 3431–3432, 2005.

[9] T. F. Oliver, B. Schmidt, and D. L. Maskell, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 12, pp. 851–855, 2005.

[10] I. T. S. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8, article 185, 2007.

[11] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "SWPS3—fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and x86/SSE2," *BMC Research Notes*, vol. 1, article 107, 2008.

[12] M. S. Farrar, "Optimizing Smith-Waterman for the Cell Broadband Engine," http://cudasw.sourceforge.net/sw-cellbe.pdf.

[13] A. Wirawan, C. K. Kwoh, N. T. Hieu, and B. Schmidt, "CBESW: sequence alignment on the playstation 3," *BMC Bioinformatics*, vol. 9, article 377, 2008.

[14] Y. Liu, W. Huang, J. Johnson, and S. H. Vaidya, "GPU accelerated Smith-Waterman," in *Computational Science—ICCS 2006*, vol. 3994 of *Lecture Notes in Computer Science*, pp. 188–195, Springer, Berlin, Germany, 2006.

[15] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, IEEE, April 2006.

[16] S. A. Manavski and G. Valle, "CUDA compatible GPU cards as efficient hardware accelerators for smith-waterman sequence alignment," *BMC Bioinformatics*, vol. 9, supplement 2, article S10, 2008.

[17] G. M. Striemer and A. Akoglu, "Sequence alignment with GPU: performance and design challenges," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, 10, p. 1, May 2009.

[18] L. Ligowski and W. Rudnicki, "An efficient implementation of Smith Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases," in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*, pp. 1–8, Rome, Italy, May 2009.

[19] Y. Liu, D. L. Maskell, and B. Schmidt, "CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units," *BMC Research Notes*, vol. 2, article 73, 2009.

[20] Y. Liu, B. Schmidt, and D. L. Maskell, "CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions," *BMC Research Notes*, vol. 3, no. 1, article 93, 2010.

[21] A. Khajeh-Saeed, S. Poole, and J. B. Perot, "Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors," *Journal of Computational Physics*, vol. 229, no. 11, pp. 4247–4258, 2010.

[22] L. Hasan, M. Kentie, and Z. Al-Ars, "GPU-accelerated protein sequence alignment," in *Proceedings of the 33rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBS '11)*, pp. 2442–2446, September 2011.

[23] D. Hains, Z. Cashero, M. Ottenberg, W. Bohm, and S. Rajopadhye, "Improving CUDASW++, a parallelization of smith-waterman for CUDA enabled devices," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium, Workshops and Phd Forum (IPDPSW '11)*, pp. 490–501, May 2011.

[24] E. F. O. Sandes and A. C. M. A. de Melo, "CUDAlign: using GPU to accelerate the comparison of megabase genomic sequences," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 137–146, January 2010.

[25] E. F. D. O. Sandes and A. C. M. A. de Melo, "Smith-Waterman alignment of huge sequences with GPU in linear space," in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, pp. 1199–1211, IEEE, Anchorage, Alaska, USA, May 2011.

[26] Y. Liu, A. Wirawan, and B. Schmidt, "CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions," *BMC Bioinformatics*, vol. 14, article 117, 2013.

[27] S.-T. Lee, C.-Y. Lin, and C. L. Hung, "GPU-based cloud service for smith-waterman algorithm using frequency distance filtration scheme," *BioMed Research International*, vol. 2013, Article ID 721738, 8 pages, 2013.

[28] S. Henikoff and J. G. Henikoff, "Amino acid substitution matrices from protein blocks," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 89, no. 22, pp. 10915–10919, 1992.

[29] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A model of evolutionary change in proteins," in *Atlas of Protein Sequence and Structure*, M. O. Dayhoff, Ed., National Biomedical Research Foundation, 1978.

[30] T. Kahveci, V. Ljosa, and A. K. Singh, "Speeding up whole-genome alignment by indexing frequency vectors," *Bioinformatics*, vol. 20, no. 13, pp. 2122–2134, 2004.

[31] T. Kahveci and A. K. Singh, "An efficient index structure for string databases," in *Proceedings of the 27th VLDB Conference*, pp. 351–360, Morgan Kaufmann, Roma, Italy, 2011.

[32] W. R. Pearson, "Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms," *Genomics*, vol. 11, no. 3, pp. 635–650, 1991.

[33] Y.-S. Lin, C.-Y. Lin, H.-C. Chi, and Y.-C. Chung, "Multiple sequence alignments with regular expression constraints on a cloud service system," *International Journal of Grid and High Performance Computing*, vol. 5, no. 3, pp. 55–64, 2013.

[34] L. Huang, C. Wu, L. Lai, and Y. Li, "Improving the mapping of Smith-Waterman sequence database searches onto CUDA-enabled GPUs," *BioMed Research International*, vol. 2015, Article ID 185179, 10 pages, 2015.

[35] X. Feng, H. Jin, R. Zheng, L. Zhu, and W. Dai, "Accelerating Smith-Waterman alignment of species-based protein sequences on GPU," *International Journal of Parallel Programming*, vol. 43, no. 3, pp. 359–380, 2015.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 259320, 10 pages
http://dx.doi.org/10.1155/2015/259320

Research Article

SimpLiFiCPM: A Simple and Lightweight Filter-Based Algorithm for Circular Pattern Matching

Md. Aashikur Rahman Azim,[1] Costas S. Iliopoulos,[2] M. Sohel Rahman,[1] and M. Samiruzzaman[2]

[1]*AℓEDA Group, Department of CSE, Bangladesh University of Engineering & Technology, Dhaka 1215, Bangladesh*
[2]*Department of Informatics, King's College London, Strand, London WC2R 2LS, UK*

Correspondence should be addressed to M. Sohel Rahman; msrahman@cse.buet.ac.bd

Received 8 February 2015; Accepted 21 March 2015

Academic Editor: Chun-Yuan Lin

This paper deals with the circular pattern matching (CPM) problem, which appears as an interesting problem in many biological contexts. CPM consists in finding all occurrences of the rotations of a pattern $\mathcal{P}$ of length m in a text $\mathcal{T}$ of length n. In this paper, we present SimpLiFiCPM (pronounced "Simplify CPM"), a simple and lightweight filter-based algorithm to solve the problem. We compare our algorithm with the state-of-the-art algorithms and the results are found to be excellent. Much of the speed of our algorithm comes from the fact that our filters are effective but extremely simple and lightweight.

1. Introduction

The classical pattern matching problem is to find all the occurrences of a given pattern $\mathcal{P}$ of length m in a text $\mathcal{T}$ of length n, both being sequences of characters drawn from a finite character set Σ. This problem is interesting as a fundamental computer science problem and is a basic requirement of many practical applications. The circular pattern, denoted by $\mathcal{C}(\mathcal{P})$, corresponding to a given pattern $\mathcal{P} = \mathcal{P}_1 \cdots \mathcal{P}_m$, is formed by connecting $\mathcal{P}_1$ with $\mathcal{P}_m$ and forming a sort of a cycle; this gives us the notion where the same circular pattern can be seen as m different linear patterns, which would all be considered equivalent. In the circular pattern matching (CPM) problem, we are interested in pattern matching between the text $\mathcal{T}$ and the circular pattern $\mathcal{C}(\mathcal{P})$ of a given pattern $\mathcal{P}$. We can view $\mathcal{C}(\mathcal{P})$ as a set of m patterns starting at positions $j \in [1 : m]$ and wrapping around the end. In other words, in CPM, we search for all "conjugates" (two words x, y are conjugate if there exist words u, v such that $x = uv$ and $y = vu$) of a given pattern in a given text.

The problem of circular pattern matching has been considered in [1], where an $\mathcal{O}(n)$-time algorithm is presented. A naive solution with quadratic complexity consists in applying a classical algorithm for searching a finite set of strings after having built the *trie* of rotations of $\mathcal{P}$. The approach presented in [1] consists in preprocessing $\mathcal{P}$ by constructing a suffix automaton of the string $\mathcal{P}\mathcal{P}$, by noting that every rotation of $\mathcal{P}$ is a factor of $\mathcal{P}\mathcal{P}$. Then, by feeding $\mathcal{T}$ into the automaton, the lengths of the longest factors of $\mathcal{P}\mathcal{P}$ occurring in $\mathcal{T}$ can be found by the links followed in the automaton in time $\mathcal{O}(n)$. In [2], the authors have presented an optimal average-case algorithm for CPM, by also showing that the average-case lower bound for the (linear) pattern matching of $\mathcal{O}(n \log_\sigma m/m)$ also holds for CPM, where $\sigma = |\Sigma|$. Recently, in [3], the authors have presented two fast average-case algorithms based on word-level parallelism. The first algorithm requires average-case time $\mathcal{O}(n \log_\sigma m/w)$, where w is the number of bits in the computer word. The second one is based on a mixture of word-level parallelism and q-grams. The authors have shown that with the addition of q-grams, and by setting $q = \mathcal{O}(\log_\sigma m)$, an optimal average-case time of $\mathcal{O}(n \log_\sigma m/m)$ can be achieved. Very recently in [4], the authors have presented an efficient algorithm for CPM that runs in $\mathcal{O}(n)$ time on average. To the best of our knowledge,

this is the fastest running algorithm for CPM in practice to date.

Notably, indexing circular patterns [5] and variations of approximate circular pattern matching under the edit distance model [6] have also been considered in the literature. Approximate circular pattern matching has also been studied recently in [4, 7]. In this paper however, we focus on the exact version of CPM.

Apart from being interesting from the pure combinatorial point of view, CPM has applications in areas like geometry, astronomy, computational biology, and so forth. For example, the following application in geometry was discussed in [5]. A polygon may be encoded spelling its coordinates. Now, given the data stream of a number of polygons, we may need to find out whether a desired polygon exists in the data stream. The difficulty in this situation lies in the fact that the same polygon may be encoded differently depending on its "starting" coordinate and hence, there exist k possible encodings where k is the number of vertices of the polygon. Therefore, instead of traditional pattern matching, we need to resort to problem CPM. This problem seems to be useful in computer graphics as well and hence may be used as a built-in function in graphics cards handling polygon rendering.

CPM in fact appears in many biological contexts. This type of circular pattern occurs in the DNA of viruses [9, 10], bacteria [11], eukaryotic cells [12], and archaea [13]. As a result, as has been noted in [14], algorithms on circular strings seem to be important in the analysis of organisms with such structures. Circular strings have also been studied in the context of sequence alignment. In [15], basic algorithms for pairwise and multiple circular sequence alignment have been presented. These results have later been improved in [16], where an additional preprocessing stage is added to speed up the execution time of the algorithm. In [17], the authors also have presented efficient algorithms for finding the optimal alignment and consensus sequence of circular sequences under the Hamming distance metric.

Furthermore, as has been mentioned in [5], this problem seems to be related to the much studied swap matching problem (in CPM, the patterns can be thought of as having a swap of two parts of it) [18] and also to the problem of pattern matching with address error (the circular pattern can be thought of as having a special type of address error) [19, 20]. For further details on the motivation and applications of this problem in computational biology and other areas the readers are kindly referred to [9–17] and references therein.

In this paper, we present SimpLiFiCPM (pronounced Simplify CPM), which is a fast and efficient algorithm for the circular pattern matching problem based on some filtering techniques. In particular, we employ a number of simple and effective filters to preprocess the given pattern and the text. After this preprocessing, we get a text of reduced length on which we can apply any existing state-of-the-art algorithms to get the occurrences of the circular pattern. So, as the name sounds, SimpLiFiCPM, in some sense, simplifies the search space of the circular pattern matching problem.

We have conducted extensive experiments to compare our algorithm with the state-of-the-art algorithms and the results are found to be excellent. Our algorithm turns out to be much faster in practice because of the huge reduction in the search space through filtering. Also, the filtering techniques we use are simple and lightweight but as can be realized from the results extremely effective.

The rest of the paper is organized as follows. Section 2 gives a preliminary description of some terminologies and concepts related to stringology that will be used throughout this paper. In Section 3 we describe our filtering algorithms. Section 4 presents the experimental results. Section 5 draws conclusion followed by some future research directions.

2. Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ϵ is a string of length 0; that is, $|\epsilon| = 0$. Let $\Sigma^+ = \Sigma^* - \{\epsilon\}$. For a string $w = xyz$, x, y, and z are called a *prefix*, *factor* (or, equivalently, *substring*), and *suffix* of w, respectively. The ith character of a string w is denoted by $w[i]$ for $1 \le i \le |w|$, and the factor of a string w that begins at position i and ends at position j is denoted by $w[i : j]$ for $1 \le i \le j \le |w|$. For convenience, we assume $w[i : j] = \epsilon$ if $j < i$. A k-factor is a factor of length k.

A circular string of length m can be viewed as a traditional linear string which has the leftmost and rightmost symbols wrapped around and stuck together in some way. Under this notion, the same circular string can be seen as m different linear strings, which would all be considered equivalent. Given a string $\mathcal{P}$ of length m, we denote by $\mathcal{P}^i = \mathcal{P}[i : m]\mathcal{P}[1 : i-1]$, $0 < i < m$, the ith *rotation* of $\mathcal{P}$ and $\mathcal{P}^0 = \mathcal{P}$.

Example 1. Suppose we have a pattern $\mathcal{P} = atcgatg$. The pattern $\mathcal{P}$ has the following rotations (i.e., conjugates): $\mathcal{P}^1 = tcgatga$, $\mathcal{P}^2 = cgatgat$, $\mathcal{P}^3 = gatgatc$, $\mathcal{P}^4 = atgatcg$, $\mathcal{P}^5 = tgatcga$, and $\mathcal{P}^6 = gatcgat$.

Here we consider the problem of finding occurrences of a pattern string $\mathcal{P}$ of length m with circular structure in a text string $\mathcal{T}$ of length n with linear structure. For instance, the DNA sequence of many viruses has a circular structure. So if a biologist wishes to find occurrences of a particular virus in a carrier's DNA sequence, which may not be circular, (s)he must locate all positions in $\mathcal{T}$ where at least one rotation of $\mathcal{P}$ occurs. This is the problem of *circular pattern matching* (CPM).

We consider the DNA alphabet, that is, $\Sigma = \{a, c, g, t\}$. In our approach, each character of the alphabet is associated with a numeric value as follows. Each character is assigned a unique number from the range $[1 \cdots |\Sigma|]$. Although this is not essential, we conveniently assign the numbers from the range $[1 \cdots |\Sigma|]$ to the characters of Σ following their inherent lexicographical order. We use $num(x), x \in \Sigma$, to denote the numeric value of the character x. So, we have $num(a) = 1$, $num(c) = 2$, $num(g) = 3$, and $num(t) = 4$. For a string S, we use the notation S_N to denote the numeric representation of the string S; $S_N[i]$ denotes the numeric value of the character $S[i]$. So, if $S[i] = g$ then $S_N[i] = num(g) = 3$. The concept of circular strings and their rotations also applies naturally on

their numeric representations as is illustrated in Example 2 below.

Example 2. Suppose we have a pattern $\mathcal{P} = atcgatg$. The numeric representation of $\mathcal{P}$ is $\mathcal{P}_N = 1423143$. And this numeric representation has the following rotations: $\mathcal{P}_N^1 = 4231431$, $\mathcal{P}_N^2 = 2314314$, $\mathcal{P}_N^3 = 3143142$, $\mathcal{P}_N^4 = 1431423$, $\mathcal{P}_N^5 = 4314231$, and $\mathcal{P}_N^6 = 3142314$.

The problem we handle in this paper can be formally defined as follows.

Problem 3 (circular pattern matching (CPM)). Given a pattern $\mathcal{P}$ of length m and a text $\mathcal{T}$ of length $n \geq m$, find all factors $\mathcal{F}$ of $\mathcal{T}$ such that $\mathcal{F} = \mathcal{P}^i$, for some $0 \leq i < m$. And if we have $\mathcal{F} = \mathcal{P}^i$ for some $0 \leq i < m$, then we say that the circular pattern $\mathcal{C}(\mathcal{P})$ matches $\mathcal{T}$ at position i.

In the context of our filter-based algorithm the concept of false positives and negatives is important. So, we briefly discuss this concept here. Suppose we have an algorithm $\mathcal{A}$ to solve a problem $\mathcal{B}$. Now suppose that $\mathcal{S}_{\text{true}}$ represents the set of true solutions for problem $\mathcal{B}$. Further suppose that $\mathcal{A}$ computes the set $\mathcal{S}_{\mathcal{A}}$ as the set of solutions for $\mathcal{B}$. Now assume that $\mathcal{S}_{\text{true}} \neq \mathcal{S}_{\mathcal{A}}$. Then, the set of false positives can be computed as follows: $\mathcal{S}_{\mathcal{A}} \setminus \mathcal{S}_{\text{true}}$, where "\" refers to the set difference operation. In other words, the set computed by $\mathcal{A}$ contains some solutions that are not true solutions for problem $\mathcal{B}$. And these are the false positives, because $\mathcal{S}_{\mathcal{A}}$ falsely marked these as solutions (i.e., positive). On the other hand, the set of false negatives can be computed as follows: $\mathcal{S}_{\text{true}} \setminus \mathcal{S}_{\mathcal{A}}$. In other words, false negatives are those members in $\mathcal{S}_{\text{true}}$ that are absent in $\mathcal{S}_{\mathcal{A}}$. These are false negatives because $\mathcal{S}_{\mathcal{A}}$ falsely marked these as nonsolutions (i.e., negative).

3. Our Approach

As has been mentioned above, our algorithm is based on some filtering techniques. Suppose we are given a pattern $\mathcal{P}$ and a text $\mathcal{T}$. We will frequently and conveniently use the expression "$\mathcal{C}(\mathcal{P})$ matches $\mathcal{T}$ at position i" (or, equivalently, "$\mathcal{P}$ circularly matches $\mathcal{T}$ at position i") to indicate that one of the conjugates of $\mathcal{P}$ matches $\mathcal{T}$ at position i. We start with a brief overview of our approach below.

3.1. Overview of SimpLiFiCPM. In SimpLiFiCPM, we first employ a number of filters to compute a set $\mathcal{N}$ of indexes of $\mathcal{T}$ such that $\mathcal{C}(\mathcal{P})$ matches $\mathcal{T}$ at position $i \in \mathcal{N}$. As will be clear shortly, our filters are unable to compute the true set of indexes and hence $\mathcal{N}$ may have false positives. However, our filters are designed in such a way that there are no false negatives. Hence, for all $j \notin \mathcal{N}$, we can be sure that there is no match. On the other hand, for all $i \in \mathcal{N}$, we may or may not have a match; that is, we may have false positives. So, after we have computed $\mathcal{N}$, we compute $\mathcal{T}'$, a reduced version of $\mathcal{T}$ concatenating all the factors $\mathcal{F}[i \cdots i + m - 1]$, $i \in \mathcal{N}$, putting a special character $\$ \notin \Sigma$ in between the factors. One essential detail is as follows. There can be i, $j \in \mathcal{N}$ such that $1 < j - i < p$. In other words, there can exist overlapping factors matching with $\mathcal{C}(P)$. However, this can be handled easily through simple bookkeeping as will be evident from our algorithm in later sections. Clearly, once we have computed the reduced text $\mathcal{T}'$ we can employ any state-of-the-art algorithm to solve CPM on $\mathcal{T}'$ to get the actual occurrences. So the most essential and useful feature of SimpLiFiCPM is the application of filters to get a reduced text on which any existing algorithm can be applied to solve CPM.

3.2. Filters of SimpLiFiCPM. In SimpLiFiCPM, we employ 6 filters. In this section we describe these filters. We also discuss the related notions and notations needed to describe these filters. In what follows we describe our filters in the context of two strings of equal length n, namely, $\mathcal{P}$ and $\mathcal{T}$, where the former is a circular string and the latter is linear. We will devise and apply different functions on these strings and present observations related to these functions which in the sequel will lead us to our desired filter. The key to our observations and the resulting filters is the fact that each function we devise results in a unique output when applied to the rotations of a circular string. For example, consider a hypothetical function $\mathcal{X}$. We will always have the relation that $\mathcal{X}(\mathcal{P}) = \mathcal{X}(\mathcal{P}^i)$ for all $1 \leq i < n$. Recall that $\mathcal{P}^0$ actually denotes $\mathcal{P}$. For the sake of conciseness, for such functions, we will abuse the notation a bit and use $\mathcal{X}(\mathcal{C}(\mathcal{P}))$ to represent $\mathcal{X}(\mathcal{P}^i)$ for all $0 \leq i < |\mathcal{P}|$.

3.2.1. Filter 1. We define the function *sum* on a string $\mathcal{P}$ of length m as follows: $sum(\mathcal{P}) = \sum_{i=1}^{m} P_N[i]$. Our first filter, Filter 1, is based on this *sum* function. We have the following observation.

Observation 1. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n. If $\mathcal{C}(\mathcal{P})$ matches $\mathcal{T}$, then we must have $sum(\mathcal{C}(\mathcal{P})) = sum(\mathcal{T})$.

Example 4. Consider $\mathcal{P} = atcgatgT = tgatcga$. As can be easily verified, here $\mathcal{P}$ circularly matches $\mathcal{T}$. In fact the match is due to the conjugate $\mathcal{P}^5$. Now we have $\mathcal{T}_N = 4314231$ and $sum(\mathcal{T}) = 18$. Then, according to Observation 1, we must have $sum(\mathcal{C}(\mathcal{P})) = 18$. This can indeed be verified easily.

Now consider another string $\mathcal{T}' = atagctg$, which is slightly different from $\mathcal{T}$. It can be easily verified that $\mathcal{C}(\mathcal{P})$ does not match $\mathcal{T}'$. Now, $\mathcal{T}'_N = 1413243$ and hence here also we have $sum(\mathcal{T}') = 18 = sum(\mathcal{C}(\mathcal{P}))$. This is an example of a false positive with respect to Filter 1.

3.2.2. Filters 2 and 3. Our second and third filters, that is, Filters 2 and 3, depend on a notion of distance between consecutive characters of a string. The *distance* between two consecutive characters of a string $\mathcal{P}$ of length m is defined by $distance(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_{\mathcal{N}}[i] - \mathcal{P}_{\mathcal{N}}[i+1]$, where $1 \leq i \leq m-1$. We define $total\_distance(P) = \sum_{i=1}^{m-1} distance(\mathcal{P}[i], \mathcal{P}[i+1])$. We also define an absolute version of it: $abs\_total\_distance(P) =$

$\sum_{i=1}^{m-1} abs(distance(\mathcal{P}[i], \mathcal{P}[i+1]))$, where $abs(x)$ returns the magnitude of x ignoring the sign. Before we apply these two functions on our strings to get our filters, we need to do a simple preprocessing on the respective string, that is, $\mathcal{P}$ in this case as follows. We extend the string $\mathcal{P}$ by concatenating the first character of $\mathcal{P}$ at its end. We use $ext(\mathcal{P})$ to denote the resultant string. So, we have $ext(\mathcal{P}) = \mathcal{P}\mathcal{P}[1]$. Since $ext(\mathcal{P})$ can simply be treated as another string, we can easily extend the notation and concept of $\mathscr{C}(\mathcal{P})$ over $ext(\mathcal{P})$ and we continue to abuse the notation a bit for the sake of conciseness as mentioned at the beginning of Section 3.2 (just before Section 3.2.1).

Now we have the following observation which is the basis of our Filter 2.

Observation 2. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathscr{C}(\mathcal{P})$ matches $\mathcal{T}$, then, we must have $abs\_total\_distance(\mathscr{C}(\mathcal{A})) = abs\_total\_distance(\mathcal{B})$. Note carefully that the function *abs_total_distance*() has been applied on the extended strings.

Example 5. Consider the same two strings of Example 4, that is, $\mathcal{P} = atcgatgT = tgatcga$. Here $\mathcal{P}$ circularly matches $\mathcal{T}$ (due to the conjugate $\mathcal{P}^5$). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $abs\_total\_distance(\mathcal{B}) = 14$. It can be easily verified that $abs\_total\_distance(\mathscr{C}(\mathcal{A}))$ is also 14.

Now consider another string $\mathcal{T}' = atagctg$ of the same length, which is slightly different from $\mathcal{T}$. It can easily be checked that $\mathscr{C}(\mathcal{P})$ does not match $\mathcal{T}'$. However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $abs\_total\_distance(\mathcal{B}')$ is still 14. So, this is an example of a false positive with respect to Filter 2.

Now we present the following related observation which is the basis of our Filter 3. Note that Observation 2 differs with Observation 3 only through using the absolute version of the function used in the latter.

Observation 3. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathscr{C}(\mathcal{P})$ matches $\mathcal{T}$, then, we must have $total\_distance(\mathscr{C}(\mathcal{A})) = total\_distance(\mathcal{B})$. Note carefully that the function *total_distance*() has been applied on the extended strings.

Example 6. Consider the same two strings of previous examples, that is, $\mathcal{P} = atcgatgT = tgatcga$. Here $\mathcal{P}$ circularly matches $\mathcal{T}$ (due to the conjugate $\mathcal{P}^5$). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $total\_distance(\mathcal{B}) = 0$. It can be easily verified that $total\_distance(\mathscr{C}(\mathcal{A}))$ is also 0.

Now consider another string $\mathcal{T}' = atagctg$ of the same length, which is slightly different from $\mathcal{T}$. It can easily be checked that $\mathscr{C}(\mathcal{P})$ does not match $\mathcal{T}'$. However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $total\_distance(\mathcal{B}')$ is still 0. So, this is an example of a false positive with respect to Filter 3.

3.2.3. Filter 4. Filter 4 uses the *sum*() function used by Filter 1, albeit in a slightly different way. In particular, it applies the *sum*() function on individual characters. So, for $x \in \Sigma$ we define $sum_x(\mathcal{P}) = \sum_{1 \le i \le |\mathcal{P}|, \mathcal{P}[i]=x} P_N[i]$. Now we have the following observation.

Observation 4. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n. If $\mathscr{C}(\mathcal{P})$ matches $\mathcal{T}$, then, we must have $sum_x(\mathscr{C}(\mathcal{P})) = sum_x(\mathcal{T})$ for all $x \in \Sigma$.

Example 7. Consider the same two strings of previous examples, that is, $\mathcal{P} = atcgatgT = tgatcga$. Recall that $\mathcal{P}$ circularly matches $\mathcal{T}$ (due to the conjugate $\mathcal{P}^5$). It is easy to calculate that $sum_a(\mathcal{T}) = 2$, $sum_c(\mathcal{T}) = 2$, $sum_g(\mathcal{T}) = 6$, and $sum_t(\mathcal{T}) = 8$. Hence according to Observation 4, the individual sum values for all the conjugates of $\mathcal{P}$ must also match this. It can be easily verified that this is indeed the case.

Now consider the other string $\mathcal{T}' = atagctg$ of the same length, which is slightly different from $\mathcal{T}$. It can easily be checked that $\mathscr{C}(\mathcal{P})$ does not match $\mathcal{T}'$. However, as we can see, still we have $sum_a(\mathcal{T}') = 2$, $sum_c(\mathcal{T}') = 2$, $sum_g(\mathcal{T}') = 6$, and $sum_t(\mathcal{T}') = 8$. This is an example of a false positive with respect to Filter 4.

Notably, a similar idea has been used by Kahveci et al. in [21] for indexing large strings with a goal to achieve fast local alignment of large genomes. In particular, for a DNA string, Kahveci et al. compute the so-called *frequency vector* that keeps track of the frequency of each character of the DNA alphabet in the string.

3.2.4. Filter 5. Filter 5 depends on modulo operation between two consecutive characters. A modulo operation between two consecutive characters of a string $\mathcal{P}$ of length m is defined as follows: $modulo(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_N[i] \% \mathcal{P}_N[i+1]$, where $1 \le i \le m-1$. We define $sum\_modulo(\mathcal{P})$ to be the summation of the results of the modulo operations on the consecutive characters of $\mathcal{P}$. More formally, $sum\_modulo(P) = \sum_{i=1}^{m-1} modulo(\mathcal{P}[i], \mathcal{P}[i+1])$. Now we present the following observation which is the basis of Filter 5. Note that this observation is applied on the extended versions of the respective strings.

Observation 5. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathscr{C}(\mathcal{P})$ matches $\mathcal{T}$, then, we must have $sum\_modulo(\mathscr{C}(\mathcal{A})) = sum\_modulo(\mathcal{B})$. Note carefully that the function *sum_modulo*() has been applied on the extended strings.

Example 8. Consider the same two strings of previous examples, that is, $\mathcal{P} = atcgatgT = tgatcga$. Recall that $\mathcal{P}$ circularly matches $\mathcal{T}$ (due to the conjugate $\mathcal{P}^5$). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $sum\_modulo(\mathcal{B}) = 5$. Now according to Observation 5, we must also have $sum\_modulo(\mathscr{C}(\mathcal{A})) = 5$. This is indeed true.

```
(1) procedure ECPS_FT(𝒫[1:m])
(2)     define five varibles for Observations 1, 2, 3, 5, and 6
(3)     define an array of size 4 for Observation 4
(4)     define an array of size 4 to keep fixed value of A, C, G, T
(5)     s ← 𝒫[1:m]𝒫[1]
(6)     initialize all defined variables to zero
(7)     initialize fixed array to {1, 2, 3, 4}
(8)     for i ← 1 to |s| do
(9)       if i ≠ |s| then
(10)          calculate different filtering values via Observations 1 and 4 and make a running sum
(11)      end if
(12)      calculate different filtering values via Observations 2, 3, 5, and 6 and make a running sum
(13)    end for
(14)    return all observations values
(15) end procedure
```

Algorithm 1: Exact circular pattern signature using Observations 1–6 in a single pass.

Now consider another string $\mathcal{T}' = tgagatc$ of the same length, which is different from $\mathcal{T}$. It can easily be checked that $\mathcal{C}(\mathcal{P})$ does not match $\mathcal{T}'$. However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $sum\_modulo(\mathcal{B}')$ is still 5. So, this is an example of a false positive with respect to Filter 5.

3.2.5. Filter 6. In Filter 6 we employ the $xor()$ operation. A bitwise exclusive-OR ($xor()$) operation between two consecutive characters of a string $\mathcal{P}$ of length m is defined as follows: $xor(\mathcal{P}[i], \mathcal{P}[i+1]) = \mathcal{P}_N[i] \wedge \mathcal{P}_N[i+1]$, where $1 \leq i \leq m-1$. We define $sum\_xor(\mathcal{P})$ to be the summation of the results of the xor operations on the consecutive characters of $\mathcal{P}$. More formally, $sum\_xor(P) = \sum_{i=1}^{m-1} xor(\mathcal{P}[i], \mathcal{P}[i+1])$. Now we present the following observation which is the basis of Filter 6. Note that this observation is applied on the extended versions of the respective strings.

Observation 6. Consider a circular string $\mathcal{P}$ and a linear string $\mathcal{T}$ both having length n and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. If $\mathcal{C}(\mathcal{P})$ matches $\mathcal{T}$, then, we must have $sum\_xor(\mathcal{C}(\mathcal{A})) = sum\_xor(\mathcal{B})$. Note carefully that the function $sum\_xor()$ has been applied on the extended strings.

Example 9. Consider the same two strings of previous examples, that is, $\mathcal{P} = atcgatgT = tgatcga$. Recall that $\mathcal{P}$ circularly matches $\mathcal{T}$ (due to the conjugate $\mathcal{P}^5$). Now consider the extended strings and assume that $\mathcal{A} = ext(\mathcal{P})$ and $\mathcal{B} = ext(\mathcal{T})$. We have $\mathcal{T}_N = 4314231$. Hence $\mathcal{B}_N = 43142314$. Hence, $sum\_xor(\mathcal{B}) = 28$. Now according to Observation 5, we must also have $sum_x or(\mathcal{C}(\mathcal{A})) = 28$. As can be verified easily, this is indeed the case.

Now consider another string $\mathcal{T}' = gtagatc$ of the same length, which is different from $\mathcal{T}$. It can easily be checked that $\mathcal{C}(\mathcal{P})$ does not match $\mathcal{T}'$. However, assuming that $\mathcal{B}' = ext(\mathcal{T}')$ we find that $sum\_xor(\mathcal{B}')$ is still 28. So, this is an example of a false positive with respect to Filter 5.

3.2.6. Discussion with respect to [8]. At this point a brief discussion with respect to our preliminary work in [8] is in order. To reduce the text $\mathcal{T}$, we also employed six filters in [8]. While Filter 1 and Filter 4 remain identical, in SimpLiFiCPM, we have changed and improved Filters 2, 3, 5, and 6 to get better results. In particular, we have introduced the concept of extended string here and modified the filters accordingly. Much of the efficiency of these new filters comes from the fact that in the preliminary version, without the extended strings, we had to deal with a set of values as the output of the functions creating a small bottleneck. On the contrary, SimpLiFiCPM now needs to deal with only one value as the output of the functions of Filters 2, 3, 5, and 6. This makes SimpLiFiCPM even faster than its predecessor. This is evident from the experimental results presented later. Notably, this has essentially brought some more changes in the overall algorithm. In particular in the searching phase of the algorithm we now need to adapt accordingly to apply the corresponding filters on the extended strings. But the overall improvement outweighs this extra work by a long margin.

3.3. Circular Pattern Signature Using the Filters. In this section, we discuss an $\mathcal{O}(m)$-time algorithm that SimpLiFiCPM uses to compute the signature of the circular pattern $\mathcal{C}(\mathcal{P})$ corresponding to pattern $\mathcal{P}$ of length m. This signature is used at a later stage to filter the text. Here, we need five variables to save the output of the functions used for Filters 1, 2, 3, 5, and 6 (based on Observations 1, 2, 3, 5, and 6). And we need a list of size 4 to save the values of the function used in Filter 4 (Observation 4). We start with the extended string $ext(\mathcal{P}) = \mathcal{P}[1 : m]\mathcal{P}[1]$ and compute the values according to Observations 1 to 6. The algorithm will iterate $m + 1$ times and hence the overall runtime of the algorithm is $\mathcal{O}(m)$. The algorithm is presented in Procedure *ECPS_FT* (Algorithm 1).

3.4. Reduction of Search Space in the Text. Now we present an $\mathcal{O}(n)$ runtime algorithm that SimpLiFiCPM uses to reduce the search space of the text applying the six filters presented above. It takes as input the pattern $\mathcal{P}[1 : m]$ of length m and the text $\mathcal{T}[1 : n]$ of length n. It calls Procedure *ECPS_FT* with $\mathcal{P}[1 : m]$ as parameter and uses the output. It then applies

```
(1) Procedure RSS_FT(T[1 : n], P[1 : m])
(2)     call ECPS_FT(P[1 : m])
(3)     save the return value of Observations 1 : 6 for further use here
(4)     define an array of size 4 to keep fixed value of A, C, G, T
(5)     initialize fixed array to {1, 2, 3, 4}
(6)     lastIndex ← 1
(7)     for i ← 1 to m do
(8)       calculate different filtering values in T[1 : m] via Observations 1–6 and make a running sum
(9)     end for
(10)    if 1–6 observations values of P[1 : m] vs 1–6 observations values of T[1 : m] have a match then
(11)                                                    ▷ Found a filtered match
(12)      Output to file T[1 : m]
(13)      lastIndex ← m
(14)    end if
(15)    for i ← 1 to n − m do
(16)      calculate different filtering values in T[1 : m] via Observations 1–6 by subtracting ith value along with
           wrapped value and adding (i + m)th value and new wrapped value to the running sum
(17)      if 1–6 filtering values of P[1 : m] vs 1–6 filtering values of T[i + 1 : i + m] have a match then
(18)                                                    ▷ Found a filtered match
(19)        if i > lastIndex then
(20)          Output an end marker $ to file
(21)        end if
(22)        if i + m > lastIndex then
(23)          if i < lastIndex then
(24)            j ← lastIndex + 1
(25)          else
(26)            j ← i + 1
(27)          end if
(28)          Output to file T[j : i + m]
(29)          lastIndex ← i + m
(30)        end if
(31)      end if
(32)    end for
(33) end procedure
```

Algorithm 2: Reduction of search space in a text string using Procedure ECPS_FT.

the same technique that is applied in Procedure *ECPS_FT* (Algorithm 1). We apply a sliding window approach with window length of m and calculate the values applying the functions according to Observations 1–6 on the factor of $\mathcal{T}$ captured by the window. Note that, for Observations 2, 3, 5, and 6, we need to consider the extended string and hence the factor of $\mathcal{T}$ within the window need be extended accordingly for calculating the values. After we calculate the values for a factor of $\mathcal{T}$, we check it against the returned values of Procedure *ECPS_FT*. If it matches, then we output the factor to a file. Note that, in case of overlapping factors (e.g., when the consecutive windows need to output the factors to a file), Procedure *ECPS_FT* outputs only the nonoverlapped characters. And Procedure *ECPS_FT* uses a \$ marker to mark the boundaries of nonconsecutive factors, where $\$ \notin \Sigma$.

Now note that we can compute the values of consecutive factors of $\mathcal{T}$ using the sliding window approach quite efficiently as follows. For the first factor, that is, $\mathcal{T}[1 \cdots m]$, we exactly follow the strategy of Procedure *ECPS_FT*. When it is done, we slide the window by one character and we only need to remove the contribution of the leftmost character of the previous window and add the contribution of the rightmost character of the new window. The functions are such that this can be done very easily using simple constant time operations. The only other issue that needs to be taken care of is due to the use of the extended string in four of the filters. But this too does not need more than simple constant time operations. Therefore, overall runtime of the algorithm is $\mathcal{O}(m) + \mathcal{O}(n - m) = \mathcal{O}(n)$. The algorithm is presented in the form of Procedure *RSS_FT* (Algorithm 2).

3.5. The Combined SimpLiFiCPM Algorithm. In this section we combine the algorithms presented so far and present the complete view of SimpLiFiCPM. We have already described the two main components of SimpLiFiCPM, namely, Procedure *ECPS_FT* and Procedure *RSS_FT*, that in fact calls the former. Now Procedure *RSS_FT* provides a reduced text $\mathcal{T}'$ (say) after filtering. At this point SimpLiFiCPM can use any algorithm that can solve CPM and apply it over $\mathcal{T}'$ and output the occurrences. Now, suppose SimpLiFiCPM uses algorithm $\mathcal{A}$ at this stage which runs in $\mathcal{O}(f(|\mathcal{T}'|))$ time. Then, clearly, the overall running time of SimpLiFiCPM is $\mathcal{O}(n) + \mathcal{O}(f(|\mathcal{T}'|))$. For example, if SimpLiFiCPM uses

Table 1: An example simulation of SimpLiFiCPM.

Iteration	Local total sum	abs sum	Actual sum	Local individual sum [0 : 4]	modulas sum	xor sum	Does it match with pattern?	Output file
1	18	14	0	{2, 2, 6, 8}	5	28	YES	*tgatcga*
2	15	12	0	{3, 2, 6, 4}	4	18	NO	$
3	13	8	0	{4, 2, 3, 4}	3	14	NO	
4	15	8	0	{3, 2, 6, 4}	6	18	NO	
5	15	8	0	{3, 2, 6, 4}	6	18	NO	
6	14	10	0	{4, 0, 6, 4}	5	18	NO	
7	12	6	0	{5, 0, 3, 4}	4	14	NO	
8	15	12	0	{4, 0, 3, 8}	5	24	NO	
9	16	12	0	{3, 2, 3, 8}	5	28	NO	
10	18	10	0	{2, 2, 6, 8}	6	24	NO	
11	16	14	0	{3, 2, 3, 8}	4	24	NO	
12	16	14	0	{3, 2, 3, 8}	4	24	NO	
13	18	14	0	{2, 2, 6, 8}	5	28	YES	*atcgatg*

the linear time algorithm of [1], then clearly the overall theoretical running time of SimpLiFiCPM will be $\mathcal{O}(n)$.

In our implementation however we have used the recent algorithm of [4], which is a linear time algorithm on average and the fastest algorithm in practice to the best of our knowledge. In particular, in [4], the authors have presented an approximate circular string matching algorithm with k-mismatches (ACSMF-Simple) via filtering. They have built a library for ACSMF-Simple algorithm. The library is freely available and can be found in [22]. In this algorithm, if we set $k = 0$, then ACSMF-Simple works for the exact matching case. In what follows, we will refer to this algorithm as ACSMF-SimpleZero*k*. We have implemented SimpLiFiCPM using ACSMF-SimpleZero*k*; that is, we have used ACSMF-Simple algorithm simply by putting $k = 0$.

3.6. An Illustrative Example. Now that we have fully described SimpLiFiCPM, in this section we present the simulation of SimpLiFiCPM on a particular example. We only show the simulation up to the output of Procedure *RSS_FT*, that is, the output of the reduced text, because afterwards we can employ any state-of-the-art algorithm within SimpLiFiCPM. Consider a pattern $\mathcal{P} = atcgatg$. The values computed by Procedure *ECPS_FT* according to Observations 1 through 6 are as follows, respectively: $local\_total\_sum = 18$, $abs\_sum = 14$, $actual\_sum = 0$, $local\_individual\_sum[0 : 4] = \{2, 2, 6, 8\}$, $modulas\_sum = 5$, and $xor\_sum = 28$.

Again consider a text string $\mathcal{T} = tgatcgaaagtaatcgatg\$$. For the first sliding window we need to calculate the observation values from $\mathcal{T}[1 : 7]$. The observation values according to Procedure *RSS_FT* are as follows for $\mathcal{T}[1 : 7]$: $local\_total\_sum = 18$, $abs\_sum = 14$, $actual\_sum = 0$, $local\_individual\_sum[0 : 4] = \{2, 2, 6, 8\}$, $modulas\_sum = 5$, and $xor\_sum = 28$.

The length of $\mathcal{T}$ is 19. And the length of $\mathcal{P}$ is 7. So, the algorithm iterates exactly $19 - 7 + 1 = 13$ times. Each iteration is illustrated in Table 1.

4. Experimental Results

We have implemented SimpLiFiCPM and conducted extensive experiments to analyze its performance. We have coded SimpLiFiCPM in C++ using a GNU compiler with General Public License (GPL). Our code is available at [23]. As has been mentioned already above, our implementation of SimpLiFiCPM uses the ACSMF-SimpleZero*k* [4]. ACSMF-Simple [4] has been implemented as library functions in the C programming language under GNU/Linux operating system. The library implementation is distributed under the GNU General Public License (GPL). It takes as input the pattern $\mathcal{P}$ of length m, the text $\mathcal{T}$ of length n, and the integer threshold $k < m$ and returns the list of starting positions of the occurrences of the rotations of $\mathcal{P}$ in $\mathcal{T}$ with k-mismatches as output. In our case we use $k = 0$.

We have used real genome data in our experiments as the text string, $\mathcal{T}$. This data has been collected from [24]. Here, we have taken 299 MB of data for our experiments. We have generated random patterns of different length by a random indexing technique in these 299 MB of text string.

We have conducted our experiments on a PowerEdge R820 rack serve PC with 6-core Intel Xeon processor *E5-4600* product family and 64 GB of RAM under GNU/Linux. With the help of the library used in [4], we have compared the running time of our preliminary work in [8] (referred to as Filter-CPM henceforth), ACSMF-SimpleZero*k* of [4], and SimpLiFiCPM. Table 2 reports the elapsed time and speed-up comparisons for various pattern sizes ($500 \leq m \leq 3000$). As can be seen from Table 2, Filter-CPM [8] runs faster than ACSMF-SimpleZero*k* in all cases. And in fact Filter-CPM [8] achieves a minimum of twofold speed-up for all the pattern sizes. Again, referring to the same table, SimpLiFiCPM runs even faster than ACSMF-SimpleZero*k* in all cases. And in fact SimpLiFiCPM achieves a minimum of threefold speed-up for all the pattern sizes.

In order to analyze and understand the effect of our filters we have run a second set of experiments as follows.

Table 2: Elapsed time (in seconds) of and speed-up comparisons among Filter-CPM [8], ACSMF-SimpleZero*k* [4], and SimpLiFiCPM on a text of size 299 MB.

m	Elapsed time (s) of ACSMF-SimpleZero*k*	Elapsed time (s) of Filter-CPM	Speed-up: ACSMF-SimpleZero*k* versus Filter-CPM	Elapsed time (s) of SimpLiFiCPM	Speed-up: ACSMF-SimpleZero*k* versus SimpLiFiCPM
500	5.938	3.025	2	1.167	5
550	7.914	3.068	3	1.456	5
600	7.691	3.06	3	1.364	6
650	7.836	3.074	3	1.006	8
700	7.739	3.072	3	1.028	8
750	7.82	3.051	3	1.073	7
800	7.839	3.209	2	1.04	8
850	8.382	3.053	3	1.055	8
900	7.646	3.055	3	1.278	6
950	7.876	3.049	3	1.402	6
1000	7.731	3.067	3	1.216	6
1600	7.334	3.206	2	1.182	6
1650	8.239	3.063	3	0.969	9
1700	7.572	3.059	2	1.18	6
1750	5.968	3.066	2	1.144	5
1800	7.551	3.064	2	1.179	6
1850	7.407	3.079	2	1.086	7
1900	7.861	3.225	2	1.126	7
1950	7.339	3.073	2	1.028	7
2000	7.814	3.062	3	1.118	7
2050	5.969	3.057	2	1.988	3
2100	5.173	3.036	2	1.187	4
2150	5.317	3.027	2	1.919	3
2200	6.032	3.168	2	1.927	3
2250	5.009	3.073	2	1.895	3
2300	5.029	3.024	2	1.891	3
2350	5.041	3.047	2	1.887	3
2400	6.036	3.046	2	1.91	3
2450	6.04	3.037	2	1.886	3
2500	7.046	3.029	2	1.976	4
2550	7.042	3.037	2	1.987	4
2600	8.043	4.029	2	2.883	3
2650	8.049	4.03	2	2.884	3
2700	8.031	4.183	2	2.892	3
2750	8.039	4.044	2	2.882	3
2800	9.026	4.067	2	2.886	3
2850	9.154	4.036	2	2.901	3
2900	10.049	4.045	2	3.134	3
2950	11.044	5.052	2	3.876	3
3000	12.044	6.039	2	3.9	3

We have run experiments on three variants of SimpLiFiCPM where the first variant (SimpLiFiCPM-$[1 \cdots 3]$) only employs Filters 1 through 3, the second variant (SimpLiFiCPM-$[1 \cdots 4]$) only employs Filters 1 through 4, and finally the third variant (SimpLiFiCPM-$[1 \cdots 5]$) employs Filters 1 through 5. Table 2 reports the elapsed time and speed-up comparisons considering various pattern sizes ($500 \leq m \leq 2000$) for ACSMF-SimpleZero*k* and the above-mentioned three variants of SimpLiFiCPM. As can be seen from Table 3, ACSMF-SimpleZero*k* is able to beat SimpLiFiCPM-$[1 \cdots 3]$

Table 3: Elapsed time (in seconds) of and speed-up comparisons among ACSMF-SimpleZero*k* and three variants of SimpLiFiCPM (considering different combination of the filters) for a text of size 299 MB.

	Filters 1 to 3			Filters 1 to 4			Filters 1 to 5		
m	Elapsed time (s) of ACSMF-SimpleZero*k*	Elapsed time (s) of SimpLiFiCPM-[1 ⋯ 3]	Speed-up: ACSMF-SimpleZero*k* versus SimpLiFiCPM-[1 ⋯ 3]	Elapsed time (s) of ACSMF-SimpleZero*k*	Elapsed time (s) of SimpLiFiCPM-[1 ⋯ 4]	Speed-up: ACSMF-SimpleZero*k* versus SimpLiFiCPM-[1 ⋯ 4]	Elapsed time (s) of ACSMF-SimpleZero*k*	Elapsed time (s) of SimpLiFiCPM-[1 ⋯ 5]	Speed-up: ACSMF-SimpleZero*k* versus SimpLiFiCPM-[1 ⋯ 5]
500	6.355	3.522	2	6.373	4.973	1	6.397	2.523	3
550	8.526	20.43	0	8.564	4.866	2	8.38	2.545	3
600	8.149	43.544	0	8.286	4.902	2	8.359	2.518	3
650	8.315	4.35	2	8.448	4.894	2	8.324	2.47	3
700	9.063	7.596	1	8.71	4.9	2	8.249	2.493	3
750	8.399	6.837	1	8.643	5.101	2	8.326	2.478	3
800	8.357	16.293	1	8.346	4.915	2	8.265	2.48	3
850	8.79	10.651	1	8.309	4.924	2	8.48	2.562	3
900	7.959	23.181	0	8.411	4.916	2	8.223	2.525	3
950	8.652	15.443	1	8.552	4.93	2	8.678	2.519	3
1000	8.285	12.399	1	8.371	4.916	2	8.375	2.616	3
1600	7.846	6.074	1	7.927	4.915	2	7.872	2.529	3
1650	8.918	2.691	3	8.878	4.904	2	8.854	2.523	4
1700	7.839	6.506	1	7.697	4.897	2	7.8	2.522	3
1750	6.252	30.173	0	6.523	5.09	1	6.399	2.526	3
1800	8.643	26.655	0	8.218	4.918	2	8.143	2.487	3
1850	8.072	2.901	3	8.026	4.901	2	8.095	2.532	3
1900	8.442	30.468	0	8.495	4.927	2	8.297	2.516	3
1950	8.123	2.542	3	8.367	4.927	2	7.951	2.495	3
2000	8.366	12.175	1	8.58	5.13	2	8.394	2.533	3

in a number of cases. However, SimpLiFiCPM-$[1 \cdots 4]$ and SimpLiFiCPM-$[1 \cdots 5]$ significantly run faster than ACSMF-SimpleZero$k$ in all cases.

5. Conclusions

In this paper, we have employed some effective lightweight filtering technique to reduce the search space of the circular pattern matching (CPM) problem. We have presented SimpLiFiCPM, an extremely fast algorithm based on the above-mentioned filters. We have conducted extensive experimental studies to show the effectiveness of SimpLiFiCPM. In our experiments, SimpLiFiCPM has achieved a minimum of threefold speed-up compared to the state-of-the-art algorithms. Much of the speed of our algorithm comes from the fact that our filters are effective but extremely simple and lightweight. The most intriguing feature of SimpLiFiCPM is perhaps its capability to plug in any algorithm to solve CPM and take advantage of it. We are now working towards adapting the filters so that it could work for the approximate version of CPM.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Part of this research has been supported by an INSPIRE Strategic Partnership Award, administered by the British Council, Bangladesh, for the project titled "Advances in Algorithms for Next Generation Biological Sequences." M. Sohel Rahman is a Commonwealth Academic Fellow funded by the UK Government who is currently on a sabbatical leave from BUET.

References

[1] M. Lothaire, *Applied Combinatorics on Words*, Cambridge University Press, New York, NY, USA, 2005.

[2] K. Fredriksson and S. Grabowski, "Average-optimal string matching," *Journal of Discrete Algorithms*, vol. 7, no. 4, pp. 579–594, 2009.

[3] K.-H. Chen, G.-S. Huang, and R.. C.-T. Lee, "Bit-parallel algorithms for exact circular string matching," *The Computer Journal*, vol. 57, no. 5, pp. 731–743, 2014.

[4] C. Barton, C. S. Iliopoulos, and S. P. Pissis, "Fast algorithms for approximate circular string matching," *Algorithms for Molecular Biology*, vol. 9, no. 1, article 9, 2014.

[5] C. S. Iliopoulos and M. S. Rahman, "Indexing circular patterns," in *WALCOM: Algorithms and Computation*, S. Nakano and M. S. Rahman, Eds., vol. 4921 of *Lecture Notes in Computer Science*, pp. 46–57, Springer, Berlin, Germany, 2008.

[6] J. Lin and D. Adjeroh, "All-against-all circular pattern matching," *Computer Journal*, vol. 55, no. 7, pp. 897–906, 2012.

[7] R. Susik, S. Grabowski, and S. Deorowicz, "Fast and simple circular pattern matching," in *Man-Machine Interactions 3: Proceedings of the 3rd International Conference on Man-Machine Interactions (ICMMI '13)*, vol. 242, pp. 537–544, Springer International Publishing, 2014.

[8] M. A. R. Azim, C. S. Iliopoulos, M. S. Rahman, and M. Samiruzzaman, "A fast and lightweight filter-based algorithm for circular pattern matching," in *Proceedings of the 5th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (BCB '14)*, pp. 621–622, ACM, Newport Beach, Calif, USA, September 2014.

[9] R. Weil and J. Vinograd, "The cyclic helix and cyclic coil forms of polyoma viral DNA," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 50, no. 4, pp. 730–738, 1963.

[10] R. Dulbecco and M. Vogt, "Evidence for a ring structure of polyoma virus DNA," *Proceedings of the National Academy of Sciences of the United States of*, vol. 50, no. 2, pp. 236–243, 1963.

[11] M. Thanbichler, S. C. Wang, and L. Shapiro, "The bacterial nucleoid: a highly organized and dynamic structure," *Journal of Cellular Biochemistry*, vol. 96, no. 3, pp. 506–521, 2005.

[12] G. Lipps, *Plasmids: Current Research and Future Trends*, Caister Academic Press, Norfolk, UK, 2008.

[13] T. Allers and M. Mevarech, "Archaeal genetics—the third way," *Nature Reviews Genetics*, vol. 6, no. 1, pp. 58–73, 2005.

[14] D. Gusfield, *Algorithms on Strings, Trees and Sequences*, Cambridge University Press, New York, NY, USA, 1997.

[15] A. Mosig, I. Hofacker, P. Stadler, and A. Zell, "Comparative analysis of cyclic sequences: viroids and other small circular RNAs," in *Proceedings of the German Conference on Bioinformatics*, vol. 83 of *Lecture Notes in Informatics*, pp. 93–102, 2006.

[16] F. Fernandes, L. Pereira, and A. T. Freitas, "CSA: an efficient algorithm to improve circular DNA multiple alignment," *BMC Bioinformatics*, vol. 10, article 230, 2009.

[17] T. Lee, J. Na, H. Park, K. Park, and J. Sim, "Finding optimal alignment and consensus of circular strings," in *Combinatorial Pattern Matching: 21st Annual Symposium, CPM 2010, New York, NY, USA, June 21–23, 2010*, vol. 6129 of *Lecture Notes in Computer Science*, pp. 310–322, Springer, Berlin, Germany, 2010.

[18] A. Amir, Y. Aumann, G. M. Landau, M. Lewenstein, and N. Lewenstein, "Pattern matching with swaps," *Journal of Algorithms*, vol. 37, no. 2, pp. 247–266, 2000.

[19] A. Amir, Y. Aumann, G. Benson et al., "Pattern matching with address errors: rearrangement distances," in *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '06)*, pp. 1221–1229, ACM, Miami, Fla, USA, January 2006.

[20] P. Ahmed, C. S. Iliopoulos, A. S. Islam, and M. S. Rahman, "The swap matching problem revisited," *Theoretical Computer Science*, vol. 557, pp. 34–49, 2014.

[21] T. Kahveci, V. Ljosa, and A. K. Singh, "Speeding up whole-genome alignment by indexing frequency vectors," *Bioinformatics*, vol. 20, no. 13, pp. 2122–2134, 2004.

[22] http://www.inf.kcl.ac.uk/research/projects/asmf/.

[23] http://teacher.buet.ac.bd/msrahman/SimpLiFiCPM/.

[24] http://hgdownload-test.cse.ucsc.edu/goldenPath/hg19/bigZips/.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 197895, 6 pages
http://dx.doi.org/10.1155/2015/197895

Research Article

A New Binning Method for Metagenomics by One-Dimensional Cellular Automata

Ying-Chih Lin[1,2]

[1]*Masters Program in Biomedical Informatics and Biomedical Engineering, Feng Chia University, No. 100, Wenhwa Road, Seatwen, Taichung 40724, Taiwan*
[2]*Department of Applied Mathematics, Feng Chia University, No. 100, Wenhwa Road, Seatwen, Taichung 40724, Taiwan*

Correspondence should be addressed to Ying-Chih Lin; yichlin@fcu.edu.tw

Received 7 January 2015; Accepted 9 February 2015

Academic Editor: Hai Jiang

More and more developed and inexpensive next-generation sequencing (NGS) technologies allow us to extract vast sequence data from a sample containing multiple species. Characterizing the taxonomic diversity for the planet-size data plays an important role in the metagenomic studies, while a crucial step for doing the study is the *binning* process to group sequence reads from similar species or taxonomic classes. The metagenomic binning remains a challenge work because of not only the various read noises but also the tremendous data volume. In this work, we propose an unsupervised binning method for NGS reads based on the one-dimensional cellular automaton (1D-CA). Our binning method facilities to reduce the memory usage because 1D-CA costs only linear space. Experiments on synthetic dataset exhibit that our method is helpful to identify species of lower abundance compared to the proposed tool.

1. Introduction

With the rapid development of next-generation sequencing (NGS) technologies, the ability to gain experimental data has far surpassed the capability to proceed with further analysis. High-throughput NGS machine is capable of sequencing millions to even billions of reads (short DNA fragments) in parallel from a sample containing many species. Within a reasonable cost, an individual laboratory can generate terabase scales of sequencing data within a day [1], which also inspires many mining tools to interpret these data [2]. Instead of traditional works for studying microbial genome on an individual bacterial strain, NGS technologies as a powerful tool greatly facilitates researchers to study the genomes of multiple microorganisms from environmental samples, while it is known as *metagenomics*. Several metagenomic projects have successfully offered valuable insights to the diverse microbial communities, such as the soil [3] and human gut [4].

An important step in metagenomic analysis is the *binning* procedure to keep together reads from similar species or taxonomic classes. There are two major methodologies for binning algorithm: supervised and unsupervised methods [5]. The former is taxonomy-dependent and similarity-based where individual reads are taxonomically grouped by aligning them to known genomes in reference databases, and subsequently reads aligned to similar genomes are grouped into bins. However, in a typical metagenomic scenario, most reads (up to 99% [6]) come from genomes of hitherto unknown organisms, which are then nonexistent in current reference databases. Taxonomy-dependent binning methods fail to identify such reads, and generally categorize them as unassigned. One alternative approach is to align the taxonomic marker genes, for example, recA, rpoB, and 16S ribosomal RNA (rRNA) [7], or particular genomic regions, for example, the internal transcribed spacer (ITS) regions [8].

As for the unsupervised method, it is taxonomy-independent and groups reads from the dataset based on the genomic signatures, such as k-mer distribution, G + C content, and codon usage [5, 9], which can be directly extracted from the nucleotide sequences. According to different signatures or observations, a number of composition-based methods are proposed as the binning tools. AbundanceBin

[10] utilizes the k-mer frequency to group reads, while TOSS [11] is based on sufficiently long mers and integrates AbundanceBin into separating reads from species with different abundances. Both fail to tackle reads from different species with similar abundance ratio [12]. The series of unsupervised binning tools of MetaCluster [12] are developed according to multiple observations, and MetaCluster 5.0 can compute the number of species shaped by the sequence reads. However, it often gives inaccurate number of species for the relatively large number of species in the dataset from the performance comparison to the binning tool MCluster [13].

On the other hand, a *cellular automaton* (CA) is a discrete computational model studied for the complex systems in mathematics, computer science, economics, biology, and so forth. It consists of a regular array of cells, with each being a finite state automaton (FSA), while the array can be in a positive number of dimensions. The state of a cell at time t is a function of the states of its neighboring cells at time $t-1$, where the function is a set of *transition rules*. One-dimensional CA considers the cells over a one-dimensional array and has been used for solving synchronization problems [14], prime generation [15], data clustering [16], real-time language recognition [17], and so on. In this work, we propose a new binning approach for NGS reads from metagenomic sequences based on one-dimensional CA by the extension of previous work [16]. Since a one-dimensional CA requires only linear memory space when running, our binning method moderates the tremendous amount of memory usage caused by NGS data. In addition, we conduct experiments to evaluate the performance and compare it with the proposed tool.

This paper is organized as follows. Section 2 introduces one-dimensional CA and our binning method step by step. Subsequently, we take the simulated dataset to assess the performance in Section 3. Finally, Section 4 draws our conclusion.

2. Binning by One-Dimensional Cellular Automaton

2.1. One-Dimensional Cellular Automaton. Cellular automata are discrete models for dynamic systems, where it was originally introduced as a computational medium for machine self-replication guided by a set of rules. The classical version of CA is based on the use of a regular array, local variables, and a function working over a neighborhood. More formally, the regular grid of CA is a set of locally interconnected FSAs that is typically placed over a regular d-dimensional lattice $\mathscr{L}$ [18]. Take the two-dimensional CA as an example, it consists of a lattice of $\mathscr{L} = m \times n$ squares called *cells*, where each is in one of a finite number of states. The *neighborhood* of a cell C is a set of topologically neighboring cells around C, and a *transition rule* ϕ applied to C defines the change of each cell in the neighborhood of C from its current state to a new one. At each iteration, the transition rule is performed on all cells. Though the number of CA applications to engineering problems is relatively few, CA has been largely involved in the simulations of complex systems [18].

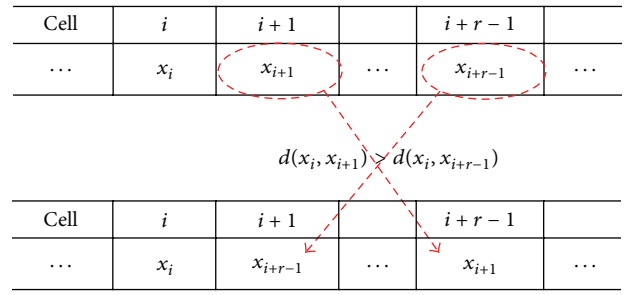


FIGURE 1: Data items x_i, x_{i+1}, and x_{i+r-1} are at cells c_i, c_{i+1}, and c_{i+r-1}, respectively. The transition rule ϕ_r applied to c_i exchanges x_{i+1} and x_{i+r-1} if $d(x_i, x_{i+1}) > d(x_i, x_{i+r-1})$.

We introduce herein a mathematical model based on the one-dimensional CA, whose cells are placed over a linear lattice $\mathscr{L}$, to describe the binning procedure for metagenomic sequences. The number of cells in the discrete lattice $\mathscr{L}$ equals the number of data items N in the dataset. At the tth iteration, each cell $c_i(t)$ for $i \in \mathscr{L}$ is the ith cell of $\mathscr{L}$ and associates with a specific item in the dataset. For a particular $r \in \{3, 4, \ldots, R\}$, applying the transition rule ϕ_r to the cell, c_i updates the neighborhood of c_i within the range r, where the parameter R can be calculated from the number of cells N. A greater value of R allows a greater size of cluster. Moreover, the boundary condition of one-dimensional CA can be periodic, fixed, or reflecting among others [18]. Here, we set the periodic manner to simulate a circular boundary; that is, $c_{N+1} = c_1$.

2.2. Transition Rules. In the beginning of $t = 0$, the data item is randomly assigned to a cell in $\mathscr{L}$, and then each cell evolves according to the function of its current state and neighboring cells, which is identified by the transition rule ϕ_r for $r \in \{3, 4, \ldots, R\}$. The value of r starts from 3 due to the minimum requirement of three neighboring cells. We say that an iteration is finished if all transition rules are performed on each cell in $\mathscr{L}$. There are two common terminated criteria to the whole process: one is the user-defined value for the maximum number of iterations and the other is the convergence of $\mathscr{L}$ to a stable state; that is, $c_i(t) = c_i(t-1), \forall i = 1, 2, \ldots, N$. We adopt the latter criterion in this work. In other words, our algorithm is terminated when there is no state change between two consecutive iterations.

At the tth iteration, let x_i be the data item at the cell c_i of $\mathscr{L}$. The transition rule ϕ_r to the cell c_i concentrates on three items of x_i, x_{i+1} and x_{i+r-1} by comparing their distances, which is the relation measurement between two items. Let $d(x_i, x_j)$ be the *distance* between two data items x_i and x_j. The rule ϕ_r applied to c_i swaps x_{i+1} and x_{i+r-1}, provided that $d(x_i, x_{i+1}) > d(x_i, x_{i+r-1})$, illustrated by Figure 1; otherwise, ϕ_r leaves the data items on c_{i+1} and c_{i+r-1} unchanged. For example, the rule ϕ_5 compares the distance $d(x_1, x_2)$ with $d(x_1, x_5)$ when applying it on c_1, with $d(x_1, x_6)$ on c_2 and so on. When ϕ_5 is applied on the cell c_{N-3}, it considers the

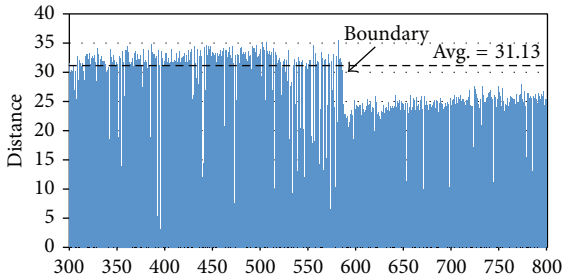


Figure 2: Chainmap diagram to our simulation result, where only the distances of cell numbers from 300 to 800 in the final state are shown. The average distance of whole dataset is 31.13, while the ideal boundary is marked at the cell c_{586}.

distances $d(x_{N-3}, x_{N-2})$ and $d(x_{N-3}, x_1)$, due to the periodic boundary condition.

To compute $d(x_i, x_j)$, two feature vectors $\vec{x}_i$ and $\vec{x}_j$ are extracted, and then we set $d(x_i, x_j) = \|\vec{x}_i - \vec{x}_j\|$. Determining the features of data is of great importance as it directly affects the measurement among data items. Typically, ideal features should be useful in distinguishing homologous patterns from the dataset, immune to noises as well as easy to extract and interpret. Such an elegant selection of features can greatly reduce the workload and simplify the subsequent clustering process. For metagenomic data, composition-based binning methods have employed several features to assign reads to different groups, where these features can be directly extracted from the nucleotide sequences including the k-mer frequency, G + C content, and codon usage [5]. In this work, we select the k-mer spectrum of a read as its feature. Previous studies suggest that $k = 4$ or 5 is the better choice to extract features from metagenomic sequences [12, 19]. As a result, we take $k = 5$ to represent each read as a 4^5-dimensional vector, and, by the transition rules, sequence reads with similar feature vectors are assembled. When the one-dimensional CA converges after a number of iterations, the reads within the same cluster would be arranged in sequence at the lattice.

2.3. Boundary Detection. The transition rules arrange the data with similar feature vectors in a chain of cells, while a problem emerges from the chain order of data: how to group the consecutive data over $\mathscr{L}$ as clusters. A feasible strategy is to identify the *boundary point*, which is the data object located in the edge of a cluster and thus may have multiple cluster features. Boundary detection of clusters plays an important role in applications, such as image processing and machine learning, but its study is still in the infancy since the first work [20]. As usual, it is challenging to accurately recognize boundary points in an efficient way due to the interference of noise and outlier points.

We detect the boundary points according to the partial order over $\mathscr{L}$ given by the one-dimensional CA. Figure 2 is the chainmap diagram of our simulation result, in which only the cell numbers from 300 to 800 in the final stationary state are shown. This diagram is composed of the distances between two items at consecutive cells, and generally the low distances of consecutive cells imply that the corresponding items should be in the same cluster. Conversely, the cells with sharp variation on distances could be candidates for the cluster boundary. It seems as if the challenge to detect boundary here can be solved by the basic sequential clustering scheme (BSAS) [21], but they are slightly different. First, the data items are arranged in a particular order retrieved from the convergence result of one-dimensional CA; second, the user-specified parameters, including the threshold of dissimilarity and maximum allowable number of clusters, as in the conventional BSAS, are not required.

It looks like there are two different clusters in Figure 2 because the distance distribution is not consistent in these cells, which inspires us to develop a straightforward way for detecting cluster boundaries. In the final state of one-dimensional CA, we consider the data item x_i at c_i. Let avg_k^i and sd_k^i be the average and standard deviation, respectively, from the starting item of the cluster k to the item x_i at c_i. Also, the average distance of the whole dataset is denoted as avg. Therefore, a item x_i is regarded as a starting point of a new cluster if the following three conditions are satisfied: (1) $d(x_i, x_{i+1}) < \text{avg}$; (2) $d(x_{i-1}, x_i) > \max(\text{avg}, \text{avg}_k^{i-1} + \text{sd}_k^{i-1})$, where x_{i-1} belongs to the cluster k; (3) $d(x_{i-1}, x_i) > d(x_i, x_{i+1}) > d(x_{i+1}, x_{i+2})$. The first criterion asks that the starting point of a new cluster should have a smaller distance than the average; the second one requests the distance between two consecutive items of a cluster boundary, that is, one is the end and the other is the start of two distinct clusters, respectively, to be bigger enough; the final criterion emphasizes the relations among consecutive items near the boundary, which is helpful to filter the noise. We use the three criteria to determine whether x_i is the starting point of a new cluster, and if yes, the item x_{i-1} is also the boundary of the preceding cluster.

2.4. Binning Method. The binning approach takes the sequencing reads from metagenomic sequences as the input, and its output is to cluster homogeneous reads together as accurate as possible. At first, our binning method assigns to each read a cell of the lattice $\mathscr{L}$ at random. Then, the one-dimensional CA with the transition rules introduced in Section 2.2 is invoked to move reads accordingly. After a number of iterations, the system converges to a stable state where all reads associated with similar feature vectors are continuously placed on $\mathscr{L}$. As long as CA arranges the reads in the chain order by composition-based comparison, the next step is to divide the chain into different slices, that is, clusters. The boundary detection given in Section 2.3 shows three criteria so as to find out the cluster boundaries. In some cases, the second condition of the three criteria for boundary detection is easy to achieve because of the small standard deviation for few items in a new cluster, resulting in excessive noises. Therefore, the value 30 is used to be the minimum size of cluster as a *rule of thumb* [22]. In other words, if x_i at the cell c_i is the starting item of a cluster, then the item located beyond the $(i + 30)$th cell can be the boundary candidate of this cluster. By this way, reads in the chain order can be divided into distinct clusters more correctly.

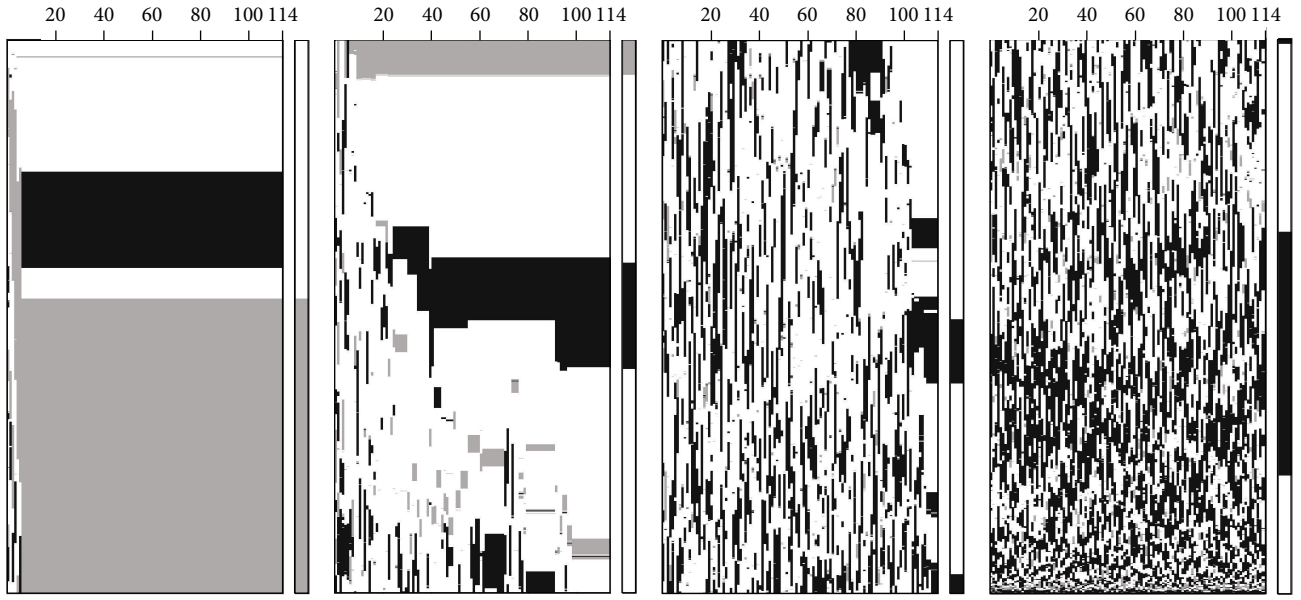


Figure 3: Experimental result of our binning method, where each row in a subdiagram represents the successive states of the corresponding cell and each column shows the cell's state in an iteration. There are totally 5,000 rows (or cells) divided into four parts, and the 1D-CA converges after 114 iterations. There are three colors associated with three species of dataset, and the rightmost bars in the four subdiagrams are the output of our binning method.

3. Experimental Results

Though there is no generally acknowledged benchmark for binning NGS reads at present, several tools are recently proposed for simulating metagenomics, such as ART [23], NeSSM [24], BEAR [25], and MetaSim [26]. Our binning method is based on the one-dimensional CA (1D-CA) to automatically determine the clusters for reads, and to validate it, the dataset as the work in [13] is generated using MetaSim. The synthetic dataset D9 in [13] consists of three species with the average length of 1,000 bp and the read number of 5,000 as well as the abundance ratio of 1 : 3 : 9. The three species in D9 are *Pseudomonas_aeruginosa_PAO*1, *Legionella_pneumophila_str._Lens*, and *Cycloclasticus_sp._P*1. In addition, we run each experiment ten times and compare the result with MCluster [13].

Figure 3 exhibits the successive states of 1D-CA. This is a grayscale image with three colors, namely, white, gray, and black, assigned to three species, respectively: white corresponds to the species of the greatest abundance, gray corresponds to the species of the smallest abundance, and black corresponds to the remained one. There are 144 iterations in total, and due to the great amount of reads, we quarter the strip of diagram as shown in Figure 3. In the beginning of the leftmost columns of the four subdiagrams, all reads are put on the lattice in a random order, leading to the mixed colors. At the 6th iteration, the leftmost subdiagram shows two distinct blocks, while the number of blocks becomes four at the end of this subdiagram. From the rightmost subdiagram in Figure 3, we can see that the reads located here are hard to be stable and thus the color distribution is in a state of utter chaos. Moreover, the rightmost bars of the four subdiagrams in Figure 3 are the clustering result of our binning method, in which it fails to identify the clear black block in the leftmost subdiagram.

In addition, we run MCluster on the same dataset and compare its performance with our method. MCluster is developed as the unsupervised method for binning metagenomic sequences, and it has been shown to be better than several works in the overall performance [13]. We run MCluster ten times as it is identical to ours and compute the average of these simulated results. To evaluate the experimental results, we consider two performance metrics, *precision* and FP *rate*. Assume that there are N_s species in the dataset and a binning algorithm identifies K clusters $C_1, C_2, \ldots, C_K$. Let R_{ij} be the number of reads in C_i that are from species j, and the cluster C_i is identified as species s if $\arg_j \max(R_{ij}) = s$. The precision can then be defined as follows [12, 13]:

$$\text{precision} = \frac{\sum_{i=1}^{K} \max_j \left(R_{ij}\right)}{\sum_{i=1}^{K} \sum_{j=1}^{N_s} R_{ij}}. \quad (1)$$

Since there is no "unclassified reads" in our binning method, two metrics *sensitivity* and *F-measure* are identical to the precision as in [13]; thus, it is excluded here. Moreover, the other metric FP *rate* is to measure the number of reads assigned to incorrect species. Let $\mathbb{C}_s$ be the index set of clusters recognized as species s. As a result, the FP *rate* of a species s, denoted as $\text{FP}_r(s)$, is defined by

$$\text{FP}_r(s) = \frac{\sum_{i \in \mathbb{C}_s} \sum_j \left\{R_{ij} \mid j = 1, 2, \ldots, s-1, s+1, \ldots, N_s\right\}}{\sum_{i \in \mathbb{C}_s} \sum_j \left\{R_{ij} \mid j = 1, 2, \ldots, N_s\right\}}. \quad (2)$$

Table 1: Performance comparison.

	1D-CA			MCluster [13]		
Precision	0.754			0.842		
s	I	II	III	I	II	III
$FP_r(s)$	0.262	0.317	0.004	0.031	0.356	0.006

Given the species s, $FP_r(s)$ measures the false positive condition to all clusters identified as s by the binning method. Table 1 summarizes the average performances of the simulated result for MCluster and ours. From this table, the 1D-CA precision is significantly lower than the MCluster precision; even so, 1D-CA has something to recommend it. To evaluate the FP *rate*, we denote the species of dataset as I, II, and III and sort them in descending order according to their abundance ratios; that is, the most abundant species corresponds to the symbol I. Table 1 shows that $FP_r(I)$ of 1D-CA is far worse than that of MCluster, implying that 1D-CA is not good at identifying abundant specsies, which is contrary to most works [10–12]. This may be caused by the insensitivity of 1D-CA to detect the cluster boundaries, and hence many reads are inaccurately recognized to the cluster of species I. However, 1D-CA has the best performance in identifying species of lower abundances as shown in Table 1, which helps in extracting rare species from samples and progressing the assembly work with ease.

4. Conclusions

Recent technologies on high-throughput NGS grow rapidly such that it is easy and cheap to sequence all individuals of a microbial community from environmental samples. Metagenomic analysis parses the vast amount of sequence data to mining valuable insights, where the binning process is a crucial step to group sequence reads from similar species or taxonomic classes. Due to the read noises and planet-size data, the binning step is challenging in metagenomic studies. In this paper, we propose a new binning method based on the one-dimensional cellular automaton (1D-CA), where 1D-CA has been used for solving synchronization problems, data clustering, language recognition, and so forth. Since 1D-CA requires only linear space when running, our method moderates the tremendous amount of memory usage resulted from NGS reads. Moreover, experiments on synthetic dataset show that our method is helpful to identify species of lower abundance compared to the proposed tool, which facilitates the recognition of rare species from environmental samples.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the cooperation project of Feng Chia University and Taichung Veterans General Hospital under Grant no. TCVGH-FCU1038204, as well as the Ministry of Science and Technology under Grant no. MOST 103-2633-S-035-001.

References

[1] F. Luciani, R. A. Bull, and A. R. Lloyd, "Next generation deep sequencing and vaccine design: today and tomorrow," *Trends in Biotechnology*, vol. 30, no. 9, pp. 443–452, 2012.

[2] Y.-C. Lin, C.-S. Yu, and Y.-J. Lin, "Enabling large-scale biomedical analysis in the cloud," *BioMed Research International*, vol. 2013, Article ID 185679, 6 pages, 2013.

[3] R. Daniel, "The metagenomics of soil," *Nature Reviews Microbiology*, vol. 3, no. 6, pp. 470–478, 2005.

[4] J. Qin, R. Li, J. Raes et al., "A human gut microbial gene catalogue established by metagenomic sequencing," *Nature*, vol. 464, no. 7285, pp. 59–65, 2010.

[5] S. S. Mande, M. H. Mohammed, and T. S. Ghosh, "Classification of metagenomic sequences: methods and challenges," *Briefings in Bioinformatics*, vol. 13, no. 6, pp. 669–681, 2012.

[6] J. A. Eisen, "Environmental shotgun sequencing: its potential and challenges for studying the hidden world of microbes," *PLoS Biology*, vol. 5, no. 3, pp. 384–388, 2007.

[7] J. R. Cole, B. Chai, R. J. Farris et al., "The ribosomal database project (RDP-II): sequences and tools for high-throughput rRNA analysis," *Nucleic Acids Research*, vol. 33, pp. D294–D296, 2005.

[8] M. Santamaria, B. Fosso, A. Consiglio et al., "Reference databases for taxonomic assignment in metagenomics," *Briefings in Bioinformatics*, vol. 13, no. 6, pp. 682–695, 2012.

[9] C. Dutta and S. Paul, "Microbial lifestyle and genome signatures," *Current Genomics*, vol. 13, no. 2, pp. 153–162, 2012.

[10] Y.-W. Wu and Y. Ye, "A novel abundance-based algorithm for binning metagenomic sequences using l-tuples," *Journal of Computational Biology*, vol. 18, no. 3, pp. 523–534, 2011.

[11] O. Tanaseichuk, J. Borneman, and T. Jiang, "Separating metagenomic short reads into genomes via clustering," *Algorithms for Molecular Biology*, vol. 7, no. 27, pp. 1–15, 2012.

[12] Y. Wang, H. C. M. Leung, S. M. Yiu, and F. Y. L. Chin, "Metacluster 5.0: a two-round binning approach for metagenomic data for low-abundance species in a noisy sample," *Bioinformatics*, vol. 28, no. 18, Article ID bts397, pp. i356–i362, 2012.

[13] R. Liao, R. Zhang, J. Guan, and S. Zhou, "A new unsupervised binning approachfor metagenomic sequences based onN-grams and automatic feature weighting," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 11, no. 1, pp. 42–54, 2014.

[14] H. Umeo, N. Kamikawa, K. Nishioka, and S. Akiguchi, "Simulation of generalized synchronization processes on one-dimensional cellular automata," in *Proceedings of the 9th WSEAS International Conference on Simulation, Modelling and Optimization (SMO '09)*, pp. 350–357, Stevens Point, Wis, USA, September 2009.

[15] P. C. Fischer, "Generation of primes by a one-dimensional real-time iterative array," *Journal of the Association for Computing Machinery*, vol. 12, pp. 388–394, 1965.

[16] J. de Lope and D. Maravall, "Data clustering using a linear cellular automata-based algorithm," *Neurocomputing*, vol. 114, pp. 86–91, 2013.

[17] A. R. Smith, "Real-time language recognition by one-dimensional cellular automata," *Journal of Computer and System Sciences*, vol. 6, no. 3, pp. 233–253, 1972.

[18] A. G. Hoekstra, J. Kroc, and P. M. A. Sloot, *Simulating Complex Systems by Cellular Automata*, Springer, 2010.

[19] B. Chor, D. Horn, N. Goldman, Y. Levy, and T. Massingham, "Genomic DNA k-mer spectra: models and modalities," *Genome Biology*, vol. 10, no. 10, article R108, 2009.

[20] C. Xia, W. Hsu, M. L. Lee, and B. C. Ooi, "BORDER: efficient computation of boundary points," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 3, pp. 289–303, 2006.

[21] S. Theodoridis and K. Koutroumbas, *Pattern Recognition*, Academic Press, 4th edition, 2008.

[22] L. Cohen, L. Manion, and K. Morrison, *Research Methods in Education*, Routledge, 7th edition, 2011.

[23] W. Huang, L. Li, J. R. Myers, and G. T. Marth, "ART: a next-generation sequencing read simulator," *Bioinformatics*, vol. 28, no. 4, Article ID btr708, pp. 593–594, 2012.

[24] B. Jia, L. Xuan, K. Cai, Z. Hu, L. Ma, and C. Wei, "NeSSM: a next-generation sequencing simulator for metagenomics," *PLoS ONE*, vol. 8, no. 10, Article ID e75448, 2013.

[25] S. Johnson, B. Trost, J. R. Long, V. Pittet, and A. Kusalik, "A better sequence-read simulator program for metagenomics," *BMC Bioinformatics*, vol. 15, supplement 9, article S14, 2014.

[26] D. C. Richter, F. Ott, A. F. Auch, R. Schmid, and D. H. Huson, "MetaSim—a sequencing simulator for genomics and metagenomics," *PLoS ONE*, vol. 3, no. 10, Article ID e3373, 2008.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 196591, 8 pages
http://dx.doi.org/10.1155/2015/196591

Research Article

Spaced Seed Data Structures for *De Novo* Assembly

Inanç Birol, Justin Chu, Hamid Mohamadi, Shaun D. Jackman, Karthika Raghavan, Benjamin P. Vandervalk, Anthony Raymond, and René L. Warren

Canada's Michael Smith Genome Sciences Centre, British Columbia Cancer Agency, Vancouver, BC, Canada V5Z 4S6

Correspondence should be addressed to Inanç Birol; ibirol@bcgsc.ca

Received 6 February 2015; Accepted 30 March 2015

Academic Editor: Che-Lun Hung

De novo assembly of the genome of a species is essential in the absence of a reference genome sequence. Many scalable assembly algorithms use the de Bruijn graph (DBG) paradigm to reconstruct genomes, where a table of subsequences of a certain length is derived from the reads, and their overlaps are analyzed to assemble sequences. Despite longer subsequences unlocking longer genomic features for assembly, associated increase in compute resources limits the practicability of DBG over other assembly archetypes already designed for longer reads. Here, we revisit the DBG paradigm to adapt it to the changing sequencing technology landscape and introduce three data structure designs for spaced seeds in the form of paired subsequences. These data structures address memory and run time constraints imposed by longer reads. We observe that when a fixed distance separates seed pairs, it provides increased sequence specificity with increased gap length. Further, we note that Bloom filters would be suitable to implicitly store spaced seeds and be tolerant to sequencing errors. Building on this concept, we describe a data structure for tracking the frequencies of observed spaced seeds. These data structure designs will have applications in genome, transcriptome and metagenome assemblies, and read error correction.

1. Introduction

For nearly a century, progressive discovery of the number and molecular structure of chromosomes and their information content have proven to be useful in the clinical domain [1, 2]. With the sequencing of the human genome, we have gained a reference for base pair resolution comparisons that have provided unprecedented insights in molecular and cellular biology. Complementing this reference, development of high throughput sequencing (HTS) platforms, most notably from Roche 454 (Basel, Switzerland), Illumina (San Diego, CA), Life Technologies (Carlsbad, CA), and Pacific Biosciences (Menlo Park, CA), significantly benefited clinical genomics [3, 4], cancer genomics in particular [5, 6]. And there is increased anticipation in the field towards a new sequencing platform from Oxford Nanopore Technologies (Cambridge, UK).

Rapid improvements in the quality and quantity of sequencing data generated by HTS platforms have called for innovative and robust bioinformatics tools. The introduction of read alignment algorithms that use advanced computing science concepts, such as Burrows-Wheeler transformation [7], Ferragina Manzini (FM) indexing [8], and cache oblivious algorithms, allowed the reference-based assembly approach to scale with the exploding volume of HTS data [9–11]. While earlier comparative genomics tools concentrated mostly on analysis of aligned reads [12, 13], the approach biased analyses toward reaffirmation of the reference, even when there is an alternative and parsimonious interpretation.

The fundamental drawback of the reference-based assembly approaches is the consideration of read data independently, ignoring that they are sampling a common underlying sequence. This becomes especially pronounced when a region is highly rearranged, expressed in an unannotated structure, or represented erroneously in the reference. To extend the utility of the reference-based assembly paradigm, several groups developed alternative alignment postprocessing approaches, such as base quality recalibration followed by realignment [14, 15], local assembly with constraints to gain base pair resolution [16–18], or developed methods that measure statistics about putative events, often foregoing base pair resolution [19, 20].

Recently, analysis of HTS data using *de novo* assembly, an approach that is unbiased by the reference sequence, is gaining interest [21–23]. Even though the approach is substantially more computationally intensive, the enhanced

specificity and the resulting savings in event verification efforts justify the choice. In earlier work, we had reported on a scalable *de novo* assembly tool, ABySS, that used short reads from an HTS platform to assemble the human genome [24], and we further demonstrated the utility of the approach to analyze transcriptome sequencing (RNA-seq) data (Trans-ABySS) [25, 26]. The technology proved to be valuable in large-scale cancer cohort studies [5, 27–29].

Sequence assembly tools differ in the way they identify read overlaps and disambiguate unclear sequence extensions. A de Bruijn graph (DBG) representation of k-mer overlaps (overlaps between sequences of k base pairs in length) was introduced with the Euler algorithm [30] and is the enabling technology behind ABySS and some of the other popular *de novo* assembly tools, such as Velvet [31].

The concept hinges on loading k-mers into the computer memory to perform fast sequence extension queries. For large target genomes and/or datasets with high sequencing error rate, memory requirement for representing k-mers might be prohibitive. ABySS solves this problem by distributing the memory load over a given number of computer nodes in a computer cluster. Minia [32] implements a Bloom filter data structure [33] to represent a DBG stochastically in small memory and navigates it using a secondary data structure.

In sequence assembly, there are several advantages of using a DBG approach compared to overlap-layout-consensus [34] or string graph based assembly algorithms [35]. The former approach uses less memory and executes faster compared to the latter two. However, with increasing read lengths in "short read" platforms like Illumina and with the gaining popularity and development of "long read" platforms like Pacific Biosciences and Oxford Nanopore, DBG based assembly algorithms need to adapt to retain their advantage.

Merely increasing the length of k-mers has several problems. For both deterministic (as in ABySS [24]) and stochastic (as in Minia [32]) representation, longer k-mers will quickly inflate the memory requirements of the assembly tools, as the experimental data will present a k-mer spectrum of increasing volume. Doing so would further result in missed sequence overlaps in the presence of mismatched bases.

The data structure reported in this paper offers a design that will be suitable for extending the utility of fast and effective DBG algorithms, hereby modifying the concept of k-mers by introduction of *spaced seeds*. As well, we describe primary and auxiliary data structures based on Bloom filters [33] with potential uses in genome, transcriptome and metagenome assemblies, and error correction.

2. Spaced Seeds

Longer reads from technological advances in sequencing platforms and sample preparation coupled with data preprocessing methods (such as the Illumina synthetic long reads) will certainly be a welcome development for assembly-based analysis. However, they also pose certain challenges, most notably due to an increase in the memory required when using longer k-mer lengths.

The maximum k-mer length that ABySS can use is a compile-time parameter. Currently, we routinely use k-mers as long as 120 bp when assembling 2×150 bp reads. However, we need to, for instance, increase the number of CPU cores we use for a typical human genome assembly from 36 cores to 48 cores when we increase the k-mer length from 96 bp to 120 bp.

Further, a DBG approach assumes that one has error-free k-mers, which becomes a strong assumption when the k-mer lengths increase, especially with the established long read technology from Pacific Biosciences typically producing reads with error rates of over 10%. Even though the Illumina platforms are generally producing good quality reads with less than 1% average error rate, with increased read lengths the probability of having an error-free k-mer still decreases geometrically.

To address both of these issues, we propose using DBGs with spaced seeds. The concept is similar to the paired DBG approach in the Pathset algorithm [36]. The difference is that assembly by spaced seeds would allow for a fixed distance between the seed pairs, as opposed to an undetermined (yet constrained) distance in the Pathset algorithm, which is sensitive to read coverage fluctuations.

For spaced seeds, we use two k-mers $[k : k]$, separated by a fixed distance, Δ. This construct is a special case of the spaced seeds that are used for read alignments [37], where a sequence to be aligned is masked by a template of "match" and "do not care" positions. The alignment process tolerates mismatches in the do not care positions, as long as the match positions agree between the query and the target. In our case, match positions are evenly split and pushed to the $3'$ and $5'$ ends of the template, and the do not care positions are collected between the flanking match positions.

We note that, as the distance between the two k-mers increases, the uniqueness of spaced seeds also increases, as demonstrated in Figure 1 for both the *E. coli* and *H. sapiens* genomes. Although this observation is anecdotal, it is compelling that the behavior is so similar for these two species with substantial phylogenetic distance. Our conjecture is that the curves we present in Figure 1 are representative of a wide variety of genomes. This allows one to use spaced seeds to achieve unique representations that are otherwise not possible with single k-mers of length $2k$, thus reducing the memory requirement for assemblies with longer reads. Such a construct would also be tolerant to sequence errors that fall within the space between the seeds.

3. Data Structures

3.1. Spaced Seeds Hash Table. A straightforward implementation of the data structure within ABySS is possible through modification of the k-mer hash table of the software. A hashed k-mer holds several pieces of information:

(1) In two-bit base encoding, the sequence content for the first observation of the k-mer or its reverse-complement: $2k$ bits.

(2) Frequency of the forward and reverse-complement observations, both maxing out at 2^{15}–30 bits (round up to 32 bits).

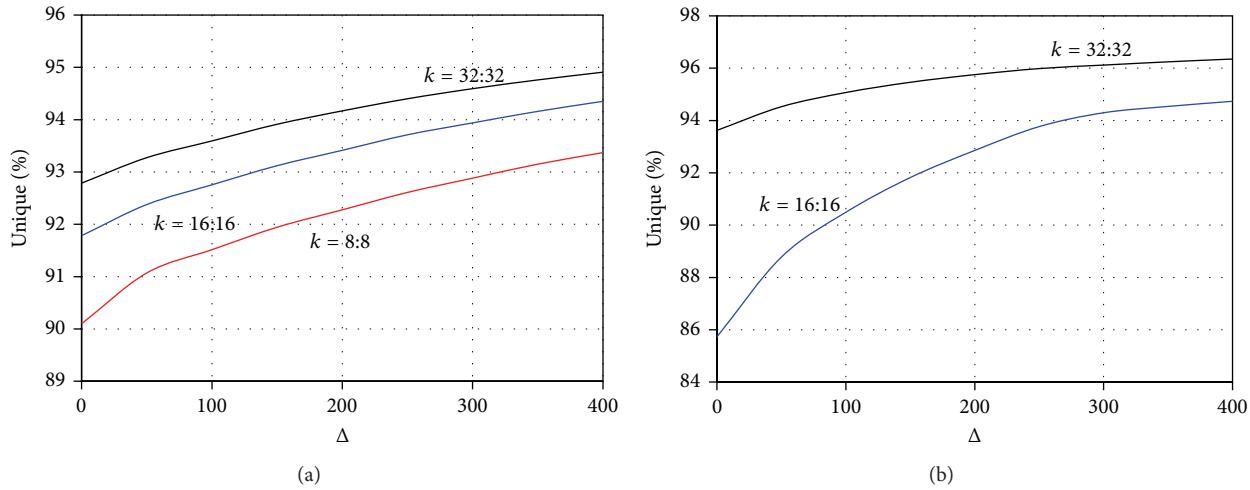


Figure 1: Uniqueness of spaced seeds in the (a) *E. coli* and (b) *H. sapiens* genomes, as a function of the space length. The red, blue, and black curves correspond to spaced seeds of lengths 8, 16, and 32 bp, respectively. When the space length is zero, the uniqueness figures correspond to 16, 32, and 64 bp single k-mer lengths, respectively. Curves show that, for the *E. coli* genome, using a spaced seeds of length 16 is equivalent to or better than using k-mers of length 64, when delta is longer than 100 bp.

(3) In the input dataset, presence (1) or absence (0) of all four possible one-base extensions of the sequence in both directions: 8 bits.

(4) Book keeping flags to track k-mers removed by error removal algorithms: 16 bits.

For a spaced seeds hash table, we modified (1) to represent the sequence content of the concatenated sequence $[k : k]$, while applying the same encoding. Of importance, the memory footprint of the spaced seeds is a constant, $4k$, for a fixed k-mer length, and does not depend on the distance between the seeds.

We also modified the sequence extension information to reflect possible extensions of either seed in either direction. This increases the memory footprint of this information to 16 bits.

The data structure for tracking spaced seed frequencies and the book keeping flags are not modified.

Overall, compared to storing a sequence of length $(2k+\Delta)$, a spaced seed represents savings in memory when $\Delta > 8$.

3.2. Spaced Seeds Bloom Filter. In the Bloom filter data structure, a number of hash functions are used to convert a sequence to a large integer value. This value specifies a unique coordinate in allocated memory, when modulus of the calculated integer is used to fit into a predefined memory size. The Bloom filter data structure offers a frugal representation of the set of reads and is typically used to query set memberships. Because of the information loss during hashing and the modulus operations, such set membership queries using a Bloom filter constructed with h hash functions holding n sequences in m bits of memory would, for *large* m, have an approximate false positive rate [38] given by

$$f = \left(1 - e^{-hn/m}\right)^h. \quad (1)$$

For a fixed target number of sequences in the filter and fixed memory size, the optimal number of hash functions can be calculated as

$$h^* = \left\lceil \frac{m}{n} \ln 2 \right\rceil. \quad (2)$$

Assuming the optimum number of hash functions to be 4, it is feasible to store the human genome in a Bloom filter with a false positive rate of $f = 6.25\%$ using ca. 2 GB of memory.

Conventional methods (e.g., CityHash from Google, Mountain View, CA) for storing h coordinates associated with a given sequence in the memory block use h different hash seeds and one hash function. (The term hash seed here refers to an initialization value for the hash function to randomize the distribution of generated hash values and should not to be confused with spaced seeds.) As an alternative, we propose the following.

Let S_{2k} be a string of length $2k$ over the alphabet $\Sigma = [A, C, G, T]$ represented using a 2-bit encoding with the alphabet mapped to $[00, 01, 10, 11]$, such that complement bases correspond through a bitwise NOT operation. Next, let sequence $S_{2k}(a : b : c)$ be a substring of S_{2k} starting from the ath letter, ending at the cth letter, sampling every bth letter, using an index origin of 1. Given a hash function, $H\{\cdot\}$, we calculate the following four hash values:

$$\begin{aligned} x_L &= H\left\{S_{2k}(1:1:k) \oplus S'_{2k}(1:1:k)\right\}, \\ x_R &= H\left\{S_{2k}(k+1:1:2k) \oplus S'_{2k}(k+1:1:2k)\right\}, \\ x_O &= H\left\{S_{2k}(1:2:2k-1) \oplus S'_{2k}(1:2:2k-1)\right\}, \\ x_E &= H\left\{S_{2k}(2:2:2k) \oplus S'_{2k}(2:2:2k)\right\}, \end{aligned} \quad (3)$$

where the substring operation is performed prior to reverse complementation denoted by "$\prime$" and $\oplus$ denotes the bitwise

Table 1: Update rules for the counting Bloom filter.

Value at bit location		Update action		
x	x'	At location	Set sign	Set count
0	0 or −0	x	0	2
Nonzero	0 or −0	x	No change	Increment
0 or −0	Nonzero	x'	1	Increment
Nonzero	Nonzero	x and x'	1	0
−0	0	x'	1	2

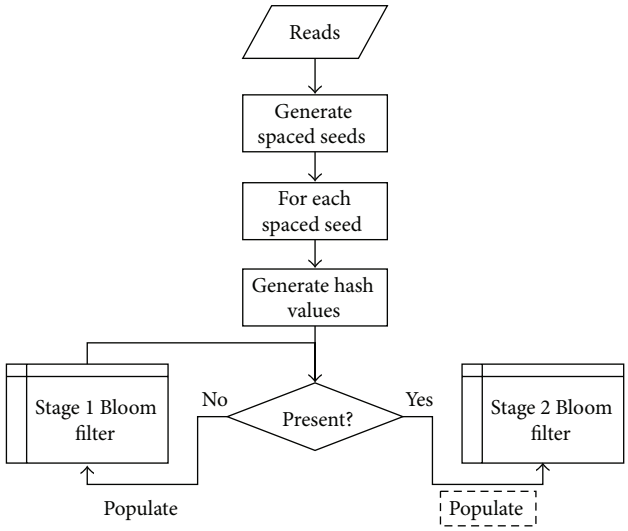


Figure 2: Flowchart of cascading Bloom filters. The process of populating the stage 2 Bloom filter, indicated by the dashed box, is described in Table 1.

XOR operation. The four values calculated in (3) can be interpreted as the left, right, odd, and even hash values for the string S_{2k}. When this string is a concatenation of two k-mers, the left and the right hash values will represent the first and the second k-mers, respectively, while the odd and even hash values will stretch through the concatenated sequence.

3.3. Counting Bloom Filter. In a sequencing experiment, it is often desirable to count the multiplicity of observed sequences. This may be valuable information for removing experimental noise in *de novo* assembly of sequencing data. In RNA-seq experiments, k-mer counts may be used to quantitate gene expression levels. Likewise, in metagenome sequencing experiments, they may be used for sequence clustering of similar k-mer coverage levels.

However, in almost all applications, the exact count does not necessarily indicate precise abundance of a sequence in the input material, as the shotgun sequencing represents a statistical sampling process. Further, if a short integer counter is used, it may saturate rapidly. On the other hand, if a long integer counter is used, it may inflate memory usage.

Building on the spaced seeds Bloom filter described above, we designed a *staged* Bloom filter with a minifloat counter, as described below.

When populating the spaced seeds Bloom filter, if the inserted sequence has already been observed (i.e., all the corresponding bits are set in the memory before populating them with the present observation) a pair of hash values are calculated for the concatenated sequence S_{2k} and its reverse-complement S'_{2k}. These hash values are collapsed by a modulus operation to point to the memory coordinates of a byte array x and x', for S_{2k} and S'_{2k}, respectively. Note that this is a deviation from the regular Bloom filters, which are *bit* arrays, as opposed to *byte* arrays. Also note that x and x' are two coordinates on the same byte array. In this construct, the "first-stage" Bloom filter of the spaced seeds records all observed S_{2k}, while the "second-stage" counting Bloom filter is engaged when it is observed at least twice (Figure 2).

At the second stage, the minifloat counter follows an IEEE 754 inspired standard to represent floating point numbers in one byte, the sign standing for the strand information of the recorded sequence. By default, we populate the counter in the direction of the first observed strand. That is, if the positions x and x' are both vacant, this would indicate that we have *cascaded* this observed sequence to the second stage for the first time. Assuming that this is not a false positive hit in our spaced seed Bloom filter, the count of observation of this sequence should be two; accordingly the position x is populated with a count of 2.

If only one of the positions is nonzero, that position is incremented by one. If it happens to be the coordinate of the reverse-complement sequence, x', then the sign bit is set to indicate that the sequence has been observed on both strands.

If both positions are nonzero, this indicates hash collision. The number negative zero is reserved as a flag for such cases, and both counts are truncated to this special flag. Note that, to avoid secondary hash collisions, that is, hash collisions due to some other pair of sequences, the nonzero condition for this case includes negative zero as well. Table 1 provides a summary of rules on how the counts are performed.

For the counts, we use a minifloat representation with one sign bit, four exponent bits, and three mantissa bits, with an exponent bias of -2 (or a 1.4.3.-2 minifloat) using the IEEE 754 standard. This gives us exact counts up to 15 and a probabilistic count beyond 16, up to a maximum of 122,880. The precision in the lower end is valuable to control for noise in experimental data. The dynamic range of over five orders of magnitude is conducive to analyzing data from RNA-seq experiments.

Table 2 illustrates some counts and their minifloat representations. Figure 3 shows the tight concordance between the approximate counts from probabilistic minifloat values and the true counts.

Table 2: Minifloat counts and their representations.

Count (c)	Mantissa[a] (t)	Exponent[a] (e)
Zeros		
0 and -0[b]	0 0 0	0 0 0 0
Subnormal numbers[c] ($c = t$)		
1	0 0 1	0 0 0 0
2	0 1 0	0 0 0 0
⋮	⋮	⋮
7	1 1 1	0 0 0 0
Normalized numbers[c] ($c = 1 \cdot t \times 2^{e+2}$)		
8	0 0 0	0 0 0 1
9	0 0 1	0 0 0 1
⋮	⋮	⋮
15	1 1 1	0 0 0 1
16	0 0 0	0 0 1 0
18	0 0 1	0 0 1 0
⋮	⋮	⋮
122,880[d]	1 1 1	1 1 1 0

[a]Most significant digits on the left.
[b]Distinguished by the sign bit.
[c]Shown for a sign bit of 0.
[d]Maximum possible 1.3.4.-2 minifloat number.

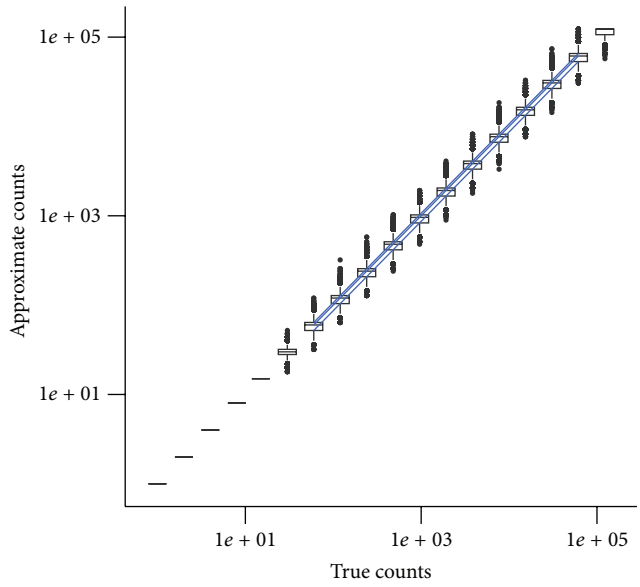


Figure 3: Approximate counts versus true counts in the minifloat data type 1.4.3.-2. The box-whisker plots indicate the interquartile range and the variability of the counts outside the first and the third quartiles. The distributions represent a repetition of 10,000 counts in each logarithmic bin.

4. Application Areas

4.1. Error Correction. Above, we noted that read errors within the space between seeds would not affect the sequence S_{2k}. Further, the Bloom filter with the four hash values x_L, x_R, x_O, and x_E would have certain characteristics when the sequencing data has read errors.

For example, when an interrogated sequence S_{2k} has a single base error, then the error should be confined to the left or the right half of the sequence. Likewise, it should be confined to the odd or even bases. Therefore, if an error-free version of the sequence is already recorded in the Bloom filter, two out of four hash values should register hits with a particular pattern, such as the left and the odd hash values. If the other two subsequences can be modified by the same one-base change, such that they also register hits, it would be a strong indication that the correct sequence should have been the sequence with this change. This can further be supported through correlating *corrected* bases to their estimated sequencing qualities, such as the q-scores generated by the data acquisition software of the sequencing instruments.

Optionally, we can use both the spaced seeds and their counts to guide error correction. Table 3 summarizes how the error correction may be performed.

4.2. Sequence Assembly. The extension of the DBG assembly algorithm using spaced seeds hash table involves minor modifications to the ABySS algorithm [24].

When the assembly of a contig is initiated, it would go through a transient phase with two "wave fronts" corresponding to extensions of the first and the second k-mer. After the seed gap is traversed, while the leading edge probes for possible extensions, the lagging edge would eliminate false branches by asserting that the extension on the lagging edge agrees with the assembled sequence between the seeds. Such assertions would realize the benefits of the improved sequence uniqueness of spaced seeds demonstrated in Figure 1.

It is also possible to use the spaced seeds Bloom filter design for sequence assembly, when the data type is supplemented with auxiliary information about the sequences of certain k-mers. While building the spaced seeds data type, one can query it at the same time for the presence or absence of 1-base extensions of the populated spaced seeds, as well as their uniqueness. These will indicate whether the k-mers under consideration are blunt ends (with no neighbors in one or both directions) or at some branching points in the corresponding DBG. If so, their sequences can be saved as auxiliary data along with their types (blunt or branch). Such sequences can be used to initiate the assembly.

During the construction of the spaced seeds Bloom filter, the list of blunt edges will go through a transitory phase, where initially there will be two blunt edges for each read. As the set of sequences converges to the target assembly size, blunt edges will be reported only at places of low or missing sequence coverage and at the edges of target sequences (as in the chromosome ends for genomes or transcript ends for transcriptomes). Depending on the size of the problem, a periodic "garbage collection" might be necessary, where the list of blunt edges is interrogated again to see if any members in the list have extensions in the populated filter and they are removed from the list if so.

However, branch sequences would not have similar issues. At branching points, it is enough to capture branching

Table 3: Error correction rules.

Value at bit location				Interpretation	Action
x_L	x_R	x_O	x_E		
1	**1**	**1**	**1**	Present in the set	Update count
0	0	0	0	Not present in the set	Insert in the filter
1	**1**	**1**	0		
1	**1**	0	**1**		
1	0	**1**	**1**		
0	**1**	**1**	**1**		
1	0	**1**	0	There may be a single base correction that would make the pattern (1 1 1 1)	If so, and if the corrected sequence has a nonzero count, correct the read. If not, insert in the filter.
1	0	0	**1**		
0	**1**	**1**	0		
0	**1**	0	**1**		
1	0	0	0	There may be two base corrections that would make the pattern (1 1 1 1)	
0	**1**	0	0		
0	0	**1**	0		
0	0	0	**1**		

using any representative of the branching sequences. While populating the filter, the first sequence to cross a branching point would not be labeled as such, but the second sequence to do so would necessarily be caught and labeled accordingly.

DBG construction using spaced seed sequences is expected to reconstruct all the edges and vertices but would also have extra edges and vertices stemming from false positive hits in the Bloom filter. However, the pattern of false positive branches on the graph would be easily distinguishable using their lengths.

If the probability of a false positive hit in our Bloom filter is f, then the probability of this branch to get extended to another false node will be f^2, and this probability will continue to drop geometrically as the length of the false branch gets longer. This represents a massive multiple hypothesis testing problem, even at the scale of the human genome where the probability of a false branch of length 10 will be very small compared to the number of hypotheses tested.

In our example of $f = 6.25\%$, the probability of a false branch of length 10, after a naïve Bonferroni correction with a factor of 3×10^9, is still less than 0.3%. Keeping in mind that the ABySS algorithm already implements a default branch trimming for branches shorter than $2k - 1$ bases long, branch removal due to false Bloom filter hits will be benign, in comparison.

By design, the spaced seeds Bloom filter harbors information at two length scales: k and $2k + \Delta$. As such, the assembly process can potentially switch between these two scales. A smaller length scale is valuable when the local sequence coverage is low, and a large length scale is valuable when the local sequence complexity is low. Being able to dynamically switch between these two length scales would potentially allow the assembly algorithm to navigate its way out of these challenging situations.

Using the coverage information captured in the counting Bloom filter data structure may further strengthen the assembly process. For example, in transcriptome assembly, the abundance of sequences in the experimental data would be indicative of the expression levels of the corresponding transcripts. Using the counting Bloom filter, one can partition the assembly problem into strata of expression levels, constructing sequences across spaced seeds with similar counts. However, the presence of alternatively spliced transcripts would be a challenge that would need to be mitigated. Allowing temporary "excursions" into other coverage strata may help resolve this issue.

Similarly, for metagenome assembly, partitioning the assembly problem into strata of counts would also be a viable approach, albeit with similar caveats. Identical or nearly identical sequences from less abundant species would yield gaps in their assembled genomes, when the higher coverage branches are used exclusively to assemble the genomes of highly abundant species. As is the case for transcriptome assemblies, count stratification can be performed taking the graph topology into account and holds the potential to disambiguate and catalog sequences with uneven representation.

5. Conclusions

The de Bruijn graph has reigned for over a decade as the data type of choice for short read assembly algorithms but could be supplanted by other paradigms as its advantages are becoming less apparent with increased read lengths. Here, we present novel data types that will help de Bruijn graphs maintain their competitive edge over alternatives. Specifically, we generalize the definition of a k-mer, the traditional workhorse of de Bruijn graphs, to a "spaced seed": a k-mer pair, consisting of two shorter sequences separated by a fixed distance, Δ.

The concept can be generalized for any topology of the spaced seed templates, such as those used for sequence alignments [37], but the special case we use is more amenable to de

Bruijn graph applications. When a spaced seed is extended within a de Bruijn graph, 1-base extensions at all the match/do not care transitions and the last matched base (if the latter corresponds to the template edge) need to be tracked. In our design, that is two bases per spaced seeds. Further, the design has the potential to be extended to represent paired end reads, when Δ is variable.

We introduce three data types to represent the spaced seeds. The first one is an extension of the k-mer hash table in the ABySS algorithm. The second one uses a specialized Bloom filter data structure to store them implicitly. Lastly, we describe the design of a counting Bloom filter in association with a minifloat counter to represent the abundance of k-mers deterministically up to 15-fold and store higher abundances in a probabilistic fashion.

Although the advantages of a Bloom filter as a low memory footprint alternative to a hash table have been reported before [32], extending the concept to include spaced seeds leverages their efficiency to report additional true positive sequences across longer distances, potentially offering longer range contiguity for *de novo* sequence assemblies. Also, the designed spaced seed data types that use Bloom filter data structures support sequence contig construction at two length scales (k and $2k + \Delta$), a feature expected to handle low coverage and sequence complexity issues. Further, we note a major advantage of the designed data types in their potential utility for sequence error correction.

Another potential use case for spaced seeds is in the problem of assembly scaffolding. We have implemented an algorithm, LINKS [39], which takes long reads, such as those generated by the Oxford Nanopore MinION sequencer to scaffold or rescaffold draft assemblies. It uses a hash table of k-mer pairs to register linkage information between assembled contigs or scaffolds. Weighing the evidence of k-mer positions on linked contigs or scaffolds and frequencies of linkage observation, it constructs paths through the linked sequences to build scaffolds.

For algorithmic complexities of various use cases, although we expect the major impact of the spaced seeds to be on reduced memory footprints, we note that reduced space complexity should result in reduced run times. That is, even if the time complexity of the algorithms as a function of the input data would not change, their scaling constants would. This behavior would be mostly due to the memory management and cache performance of the affected algorithms.

The spaced seed data type concepts proposed herein will help us retain the succinct representation advantages of de Bruijn graphs for sequence assembly of data from high throughput sequencing technologies and carry the paradigm forward to the era of long reads, when it arrives.

Disclaimer

The content is solely the responsibility of the authors and does not necessarily represent the official views of any of their funders.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported in part by funding from Genome Canada, Genome British Columbia, and British Columbia Cancer Foundation. Additional funding was provided by the National Human Genome Research Institute of the National Institutes of Health under Award no. R01HG007182.

References

[1] L. Decourt, A. J. Gebara, M. C. Lima, D. Delascio, E. Chiorboli, and J. M. Fernandes, "Turner syndrome: presentation of four cases," *Revista Paulista de Medicina*, vol. 45, no. 2, pp. 251–264, 1954.

[2] S. Levi, "Nature and origin of mongolism; critical review of etiopathogenetic problem," *Rivista di Clinica Pediatrica*, vol. 49, no. 3, pp. 171–186, 1951.

[3] L. G. Biesecker, J. C. Mullikin, F. M. Facio et al., "The ClinSeq Project: piloting large-scale genome sequencing for research in genomic medicine," *Genome Research*, vol. 19, no. 9, pp. 1665–1674, 2009.

[4] K. V. Voelkerding and E. Lyon, "Digital fetal aneuploidy diagnosis by next-generation sequencing," *Clinical Chemistry*, vol. 56, no. 3, pp. 336–338, 2010.

[5] R. D. Morin, M. Mendez-Lago, A. J. Mungall et al., "Frequent mutation of histone-modifying genes in non-Hodgkin lymphoma," *Nature*, vol. 476, no. 7360, pp. 298–303, 2011.

[6] S. P. Shah, A. Roth, R. Goya et al., "The clonal and mutational evolution spectrum of primary triple-negative breast cancers," *Nature*, vol. 486, no. 7403, pp. 395–399, 2012.

[7] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Tech. Rep. 124, Digital Equipment Corporation, 1994.

[8] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Proceedings of the 41st Annual Symposium on Foundations of Computer SCIence (Redondo Beach, CA, 2000)*, pp. 390–398, IEEE Comput. Soc. Press, Los Alamitos, Calif, USA, 2000.

[9] F. Hach, F. Hormozdiari, C. Alkan, I. Birol, E. E. Eichler, and S. C. Sahinalp, "MrsFAST: a cache-oblivious algorithm for short-read mapping," *Nature Methods*, vol. 7, no. 8, pp. 576–577, 2010.

[10] B. Langmead and S. L. Salzberg, "Fast gapped-read alignment with Bowtie 2," *Nature Methods*, vol. 9, no. 4, pp. 357–359, 2012.

[11] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[12] L. W. Hillier, G. T. Marth, A. R. Quinlan et al., "Whole-genome sequencing and variant discovery in *C. elegans*," *Nature Methods*, vol. 5, no. 2, pp. 183–188, 2008.

[13] H. Li, J. Ruan, and R. Durbin, "Mapping short DNA sequencing reads and calling variants using mapping quality scores," *Genome Research*, vol. 18, no. 11, pp. 1851–1858, 2008.

[14] C. A. Albers, G. Lunter, D. G. MacArthur, G. McVean, W. H. Ouwehand, and R. Durbin, "Dindel: accurate indel calls from short-read data," *Genome Research*, vol. 21, no. 6, pp. 961–973, 2011.

[15] A. McKenna, M. Hanna, E. Banks et al., "The genome analysis toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[16] P. Carnevali, J. Baccash, A. L. Halpern et al., "Computational techniques for human genome resequencing using mated gapped reads," *Journal of Computational Biology*, vol. 19, no. 3, pp. 279–292, 2012.

[17] K. Chen, J. W. Wallis, C. Kandoth et al., "Breakfusion: targeted assembly-based identification of gene fusions in whole transcriptome paired-end sequencing data," *Bioinformatics*, vol. 28, no. 14, pp. 1923–1924, 2012.

[18] K. Wong, T. M. Keane, J. Stalker, and D. J. Adams, "Enhanced structural variant and breakpoint detection using SVMerge by integration of multiple detection methods and local assembly," *Genome Biology*, vol. 11, no. 12, p. R128, 2010.

[19] K. Chen, J. W. Wallis, M. D. McLellan et al., "BreakDancer: an algorithm for high-resolution mapping of genomic structural variation," *Nature Methods*, vol. 6, no. 9, pp. 677–681, 2009.

[20] A. McPherson, F. Hormozdiari, A. Zayed et al., "deFuse: an algorithm for gene fusion discovery in tumor RNA-Seq data," *PLOS Computational Biology*, vol. 7, no. 5, Article ID e1001138, 2011.

[21] M. G. Grabherr, B. J. Haas, M. Yassour et al., "Full-length transcriptome assembly from RNA-Seq data without a reference genome," *Nature Biotechnology*, vol. 29, no. 7, pp. 644–652, 2011.

[22] H. Li, "Exploring single-sample snp and indel calling with whole-genome de novo assembly," *Bioinformatics*, vol. 28, no. 14, Article ID bts280, pp. 1838–1844, 2012.

[23] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney, "Oases: robust de novo RNA-seq assembly across the dynamic range of expression levels," *Bioinformatics*, vol. 28, no. 8, Article ID bts094, pp. 1086–1092, 2012.

[24] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. M. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, 2009.

[25] I. Birol, S. D. Jackman, C. B. Nielsen et al., "De novo transcriptome assembly with ABySS," *Bioinformatics*, vol. 25, no. 21, pp. 2872–2877, 2009.

[26] G. Robertson, J. Schein, R. Chiu et al., "*De novo* assembly and analysis of RNA-seq data," *Nature Methods*, vol. 7, no. 11, pp. 909–912, 2010.

[27] T. J. Pugh, O. Morozova, E. F. Attiyeh et al., "The genetic landscape of high-risk neuroblastoma," *Nature Genetics*, vol. 45, no. 3, pp. 279–284, 2013.

[28] K. G. Roberts, R. D. Morin, J. Zhang et al., "Genetic alterations activating kinase and cytokine receptor signaling in high-risk acute lymphoblastic leukemia," *Cancer Cell*, vol. 22, no. 2, pp. 153–166, 2012.

[29] S. Yip, Y. S. Butterfield, O. Morozova et al., "Concurrent CIC mutations, IDH mutations, and 1p/19q loss distinguish oligodendrogliomas from other cancers," *Journal of Pathology*, vol. 226, no. 1, pp. 7–16, 2012.

[30] P. A. Pevzner and H. Tang, "Fragment assembly with double-barreled data," *Bioinformatics*, vol. 17, no. 1, pp. S225–S233, 2001.

[31] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[32] R. Chikhi and G. Rizk, "Space-efficient and exact de Bruijn graph representation based on a Bloom filter," *Algorithms for Molecular Biology*, vol. 8, article 22, 2013.

[33] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[34] X. Huang and A. Madan, "CAP3: a DNA sequence assembly program," *Genome Research*, vol. 9, no. 9, pp. 868–877, 1999.

[35] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Research*, vol. 22, no. 3, pp. 549–556, 2012.

[36] S. K. Pham, D. Antipov, A. Sirotkin, G. Tesler, P. A. Pevzner, and M. A. Alekseyev, "Pathset graphs: a novel approach for comprehensive utilization of paired reads in genome assembly," *Journal of Computational Biology*, vol. 20, no. 4, pp. 359–371, 2013.

[37] L. Ilie, S. Ilie, and A. M. Bigvand, "SpEED: fast computation of sensitive spaced seeds," *Bioinformatics*, vol. 27, no. 17, pp. 2433–2434, 2011.

[38] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: a survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[39] R. L. Warren, B. P. Vandervalk, S. J. M. Jones, and I. Birol, "LINKS: scaffolding genome assemblies with kilobase-long nanopore reads," *bioRxiv*, 2015.

Hindawi Publishing Corporation
International Journal of Genomics
Volume 2015, Article ID 950905, 9 pages
http://dx.doi.org/10.1155/2015/950905

Research Article

Accelerating Multiple Compound Comparison Using LINGO-Based Load-Balancing Strategies on Multi-GPUs

Chun-Yuan Lin,[1] Chung-Hung Wang,[1] Che-Lun Hung,[2] and Yu-Shiang Lin[3]

[1]*Department of Computer Science and Information Engineering, Chang Gung University, Taoyuan 33302, Taiwan*
[2]*Department of Computer Science and Communication Engineering, Providence University, Taichung 43301, Taiwan*
[3]*Department of Computer Science, National Tsing Hua University, Hsinchu 30013, Taiwan*

Correspondence should be addressed to Che-Lun Hung; clhung@pu.edu.tw

Received 19 March 2015; Accepted 2 September 2015

Academic Editor: Hai Jiang

Compound comparison is an important task for the computational chemistry. By the comparison results, potential inhibitors can be found and then used for the pharmacy experiments. The time complexity of a pairwise compound comparison is $O(n^2)$, where n is the maximal length of compounds. In general, the length of compounds is tens to hundreds, and the computation time is small. However, more and more compounds have been synthesized and extracted now, even more than tens of millions. Therefore, it still will be time-consuming when comparing with a large amount of compounds (seen as a multiple compound comparison problem, abbreviated to MCC). The intrinsic time complexity of MCC problem is $O(k^2n^2)$ with k compounds of maximal length n. In this paper, we propose a GPU-based algorithm for MCC problem, called CUDA-MCC, on single- and multi-GPUs. Four LINGO-based load-balancing strategies are considered in CUDA-MCC in order to accelerate the computation speed among thread blocks on GPUs. CUDA-MCC was implemented by C+OpenMP+CUDA. CUDA-MCC achieved 45 times and 391 times faster than its CPU version on a single NVIDIA Tesla K20m GPU card and a dual-NVIDIA Tesla K20m GPU card, respectively, under the experimental results.

1. Introduction

A new drug to market usually costs a lot of time for the research and the development and invests a huge amount of money. After decoding the human genome, the molecular biology and the proteomic fields make a remarkable advance; people understand more clearly the disease generation and the disease mechanism. Computer-Aided Drug Design (abbreviated to CADD) [1] becomes an emerging research field, and it is helpful to improve the efficiencies of drug design and development. CADD is a kind of approaches, named rational drug design. Rational drug design is based on the structure, the property, and the mechanism. There are two major approaches: structure-based approach and ligand-based approach. Researchers use these two approaches to develop drugs depending on different drug design strategies. Structure-based approach mainly uses the techniques of docking [2, 3] and de novo ligand design [4], and the ligand-based approach uses the techniques of QSAR [5] and Pharmacophore [3, 6] mostly. Based on these two approaches, potential inhibitors can be found for the target genes or proteins. After that, in general, these potential inhibitors will be used to compare with the compound databases, such as ZINC [7], PubChem [8], and GDB-13 [9], in order to find other compounds with similar structure. These similar compounds may be helpful for shortening the subsequent synthesis procedures of potential inhibitors. The potential inhibitors and similar compounds are then used for the pharmacy experiments to test the biological activity, toxicity, and so forth. For well-known drugs, the compound comparison also can be used to find the generic drugs.

Therefore, compound comparison has been an important and commonly used task for the computational chemistry. Many algorithms [10] have been proposed to do the compound comparisons in the past. The Tanimoto coefficient is one of the most popular measurements between two molecules (compounds) due to its computational efficiency and its relevance to biological profile [11, 12]. In most of

the works, molecules can be represented as fingerprints and SMILES, and then the work of compound comparison can be seen as the string comparison problem. For example, the LINGO method [13] proposed by Vidal et al. is a simple algorithm to compute the chemical similarity between two SMILES-based molecules, and it has demonstrated the accuracy comparable to fingerprint methods [14]. For a pairwise compound comparison, the time complexity is $O(n^2)$, where n is the maximal length of compounds. In general, the length of compound is short (e.g., tens to hundreds) and the computation time is small. However, it will be time-consuming when compared with a large amount of compounds (denoted by a multiple compound comparison problem, abbreviated to MCC), such as ZINC database with more than 60 million compounds and GDB-13 database with more than 970 million small molecules. The intrinsic time complexity of MCC problem is $O(k^2n^2)$ with k compounds of maximal length n. Hence, how to accelerate the MCC problem is an important issue.

It is a feasible direction to apply parallel technologies and multicore devices into the above issue. The feasibility of using massive computational devices to enhance the performance of many programs has received considerable attention in recent years, especially for many-core devices, such as FPGAs [15–17] and Cell/Bes [18–20]. Current high-end graphics processing units (abbreviated to GPUs) [21, 22], which contain up to thousands of cores per chip, are widely used in the high performance computing community. As a massively multithreaded processor, GPU expects the thousands of concurrent threads to fully utilize its computing power. The ease of accessing GPUs by using General-Purpose computing on Graphics Processing Units (abbreviated to GPGPU) such as Open Computing Language (abbreviated to OpenCL, https://www.khronos.org/opencl/) and compute unified device architecture (abbreviated to CUDA [23]), as opposed to graphic APIs (as OpenGL), has made the supercomputing available widely. CUDA uses a new computing architecture referred to as single instruction multiple threads (SIMT), which differs from Flynn's classification [24]. Importantly, the computing power and the memory bandwidth for modern GPUs have made porting applications more possible.

For the MCC problem, it can be done to compare a compound with a set of compounds (denoted by one to all, abbreviated to O2A) or to compare two sets of compounds (denoted by all to all, abbreviated to A2A). Several GPU-based parallel algorithms were proposed in the past. For example, Haque et al. proposed a GPU-based parallel algorithm, called SIML (full name is Single-Instruction Multiple-LINGO [25]), to calculate the Tanimoto coefficients between SMILES-based molecules. The SIML algorithm is designed based on the LINGO method, and its GPU implementation is over 30 times faster than its CPU version. Ma et al. presented a parallel algorithm [26] to calculate the Tanimoto coefficients for MCC problem between molecular fingerprints on GPUs. The experimental results showed that the implemented program achieved 39 times faster than Sybyl Database Comparison program that runs on CPUs and 10 times faster than other GPU-based programs [25, 27]. However, it is unfair to compare their algorithm based on the fingerprints representation with other algorithms based on the SMILES representation. The reason is that the computation is different for these two representations. Among these works, it is still insufficient for designing a GPU-based algorithm for MCC problem. At first, most of these works were focused on single-GPU card. Second, their tests were based on old GPU cards and CUDA capability. Third, they did not apply various load-balancing strategies into their works and then discuss their effects. A suitable load-balancing strategy can accelerate the computation speed on single- and multi-GPU cards.

Hence, in this paper, we propose a GPU-based algorithm for MCC problem (O2A and A2A) on single- and multi-GPUs, called CUDA-MCC. As the work [25], CUDA-MCC is also based on the LINGO method to calculate the Tanimoto coefficients between SMILES-based molecules. Four LINGO-based load-balancing strategies were also applied into CUDA-MCC by considering the LINGO score, LINGO number, LINGO length, and LINGO magnitude, respectively. CUDA-MCC was implemented by C+OpenMP+CUDA for single- and multi-GPU cards. The experimental tests were done on single NVIDIA Tesla K20m GPU card and dual-NVIDIA Tesla K20m GPU cards, and the experimental results showed that CUDA-MCC can achieve 45 times and 391 times faster than its CPU version on the above experimental environments, respectively.

2. Background Knowledge

The LINGO method is to model a molecule as a collection of substrings of SMILES representation (seen as a string). Thus, a SMILES string is fragmented into all substrings with length of q by using the sliding window scheme. These substrings are stored as a set of q-Lingos. In addition, the information of each q-Lingo is also stored in order to do the following comparison. For example, as shown in the literature [25], the score, length, number, and magnitude of each q-Lingo are stored. When doing the pairwise compound comparison, two sets of q-Lingos from two molecules are used to find the same Lingos. The number of the same Lingos is then used to calculate the Tanimoto coefficients. The details of LINGO method can be found in the literature [13]. Therefore, there are four LINGO-based load-balancing strategies by considering the score, length, number, and magnitude of each molecule represented by q-Lingos.

CUDA is an extension of commonly used programming languages, such as C/C++, in which users can write scalable multithreading programs for various applications. In general, the CUDA program is implemented in two parts: host and device. The host part is executed on CPU, and the device part is executed on GPU. The function executed on the device part is called a kernel. The kernel can be invoked as a set of concurrently executing threads. These threads are grouped into a hierarchical organization which can be combined into thread blocks and grids. A grid is a set of independent thread blocks, and a thread block contains many threads. The size of

grid is the number of thread blocks per grid, and the size of thread block is the number of threads per thread block.

Threads in a thread block can communicate and synchronize with each other. Threads within a thread block can communicate through a per thread block shared memory, whereas threads in the different thread blocks fail to communicate or synchronize directly. Besides shared memory, five memory types are per grid private local memory, global memory for data shared by all thread blocks, texture memory, constant memory, and registers. Of these memory types, constant memory and texture memory can be regarded as fast read only caches; the fastest memories are the registers and shared memory. The global memory, local memory, texture memory, and constant memory are located on the GPU's memory. Besides shared memory accessed by a single thread block and register only accessed by a single thread, the other memory types can be used by all of the threads. The caches of texture memory and constant memory are limited to 8 KB per streaming multiprocessor. The optimum access strategy for constant memory is all threads reading the same memory address. The cache of texture memory is designed for threads to read through the proximity of the address in order to achieve an improved reading efficiency. Fermi and Kepler architectures have real configurable L1 per streaming multiprocessor and unified L2 caches among streaming multiprocessors. Hence, L2 caches can be accessed by global memory and each streaming multiprocessor can use the L1 caches and shared memory.

The basic processing unit in NVIDIA's GPU architecture is called the streaming processor. In Fermi and Kepler architectures, the basic processing unit is called CUDA cores. Many streaming processors perform the computations on GPU. Several streaming processors can be integrated into a streaming multiprocessor according to various architectures, such as 32 and 192 streaming processors per streaming multiprocessor for Fermi and Kepler architectures, respectively. While the program runs the kernel, the device schedules thread blocks for the execution on the streaming multiprocessor. The SIMT scheme refers to threads running on the streaming multiprocessor in a small group of 32, called a warp. The warp scheduler simultaneously schedules and dispatches instructions.

3. Method

In CUDA-MCC, the goal is to compare two sets of compounds (A2A) listed as *Query* and *Database* at first and then find the compounds in *Database* with more than 0.85 Tanimoto coefficients for each compound in *Query*. CUDA-MCC also can be used to do the O2A comparison when the *Query* is with only one compound. For each compound in *Query* and *Database*, it should be fragmented into a set of q-Lingos mentioned in Section 2, respectively. Grant et al. [14] have demonstrated that setting $q = 4$ can have the best performance in various cheminformatics applications. Hence, in CUDA-MCC, the q is set to 4. Since this procedure is only done once, a *preprocessing phase* is designed in CUDA-MCC to do this procedure on CPU. After this phase, the information of *Query* and *Database* is transferred from CPU to GPU, and then a GPU implementation of *comparison phase* is designed in CUDA-MCC in order to accelerate the computation speed. Finally, the Tanimoto coefficient of each pair of compounds is stored in a result array on GPU, and then this result array will be transferred from GPU to CPU. All the compounds in *Database* with more than 0.85 Tanimoto coefficients for each compound in *Query* are reported in the *output phase* on CPU. The flowchart of three phases in CUDA-MCC is shown in Figure 1. In Figure 1, the first three processes for *Query* and *Database* are the *preprocessing phase*, followed by the *comparison phase*, and *output phase* which is the last phase. The details of these three phases are described below.

3.1. Preprocessing Phase. This phase can be divided into three parts: *reading files*, *Lingo construction,* and *Lingo sorting*. In the *reading files* part, there are two databases *Query* and *Database,* as input files should be read from the disk to memory space on CPU. For these two databases, the compounds are stored in two-dimensional string arrays, Q and Db, respectively. In Q and Db, each compound is represented as the SMILES code, as shown in Figure 2.

After that, in the *Lingo construction* part, each SMILES code (as a string) is fragmented into a set of 4-Lingos (as substrings) by using the sliding window scheme with an offset 1 on CPU. For a 4-Lingo, it will be transformed into a 32-bit integer (called **LINGO score**) according to the ASCII code table in order to accelerate the comparison in the *comparison phase*. For a SMILES code with length l, it can be fragmented into l-3 4-Lingos and this value l is called **LINGO length**. Hence, for a SMILES code, a temporary one-dimensional integer array and an integer variable are used to store the LINGO score of each 4-Lingo and LINGO length, respectively. Since there are possible repeats (for each 4-Lingo) in this array, the number of repeats for each 4-Lingo is calculated. An integer variable, **LINGO number**, for each LINGO score, is used to record the times of a 4-Lingo appearing in this compound; for example, for a 4-Lingo, 1 represents only once and two or more represents the repeats. The number of 4-Lingos without repeats is also calculated and then recorded in an integer variable, **LINGO magnitude**. For a compound, there are four LINGO types: LINGO scores, LINGO length, LINGO numbers, and LINGO magnitude. An example of a SMILES code in the *preprocessing phase* is shown in Figure 2. In order to simplify the figure, the remaining LINGO scores of nine 4-Lingos are omitted in Figure 2. In this case, the LINGO number is 1 for each 4-Lingo. The LINGO length is 13 and the LINGO magnitude is 10 which is equal to the number of 4-Lingos.

For a pair of compounds, the Tanimoto coefficient is calculated according to the number of similar 4-Lingos. Therefore, in order to accelerate the computation and reduce the unnecessary comparisons in the *comparison phase*, for each compound, its LINGO scores are sorted by using the quick sort algorithm on CPU in the *Lingo sorting* part. Therefore, for *Query* and *Database*, a two-dimensional integer array is used to store the sorted LINGO scores of each compound (4-Lingos); a two-dimensional integer array is used to store

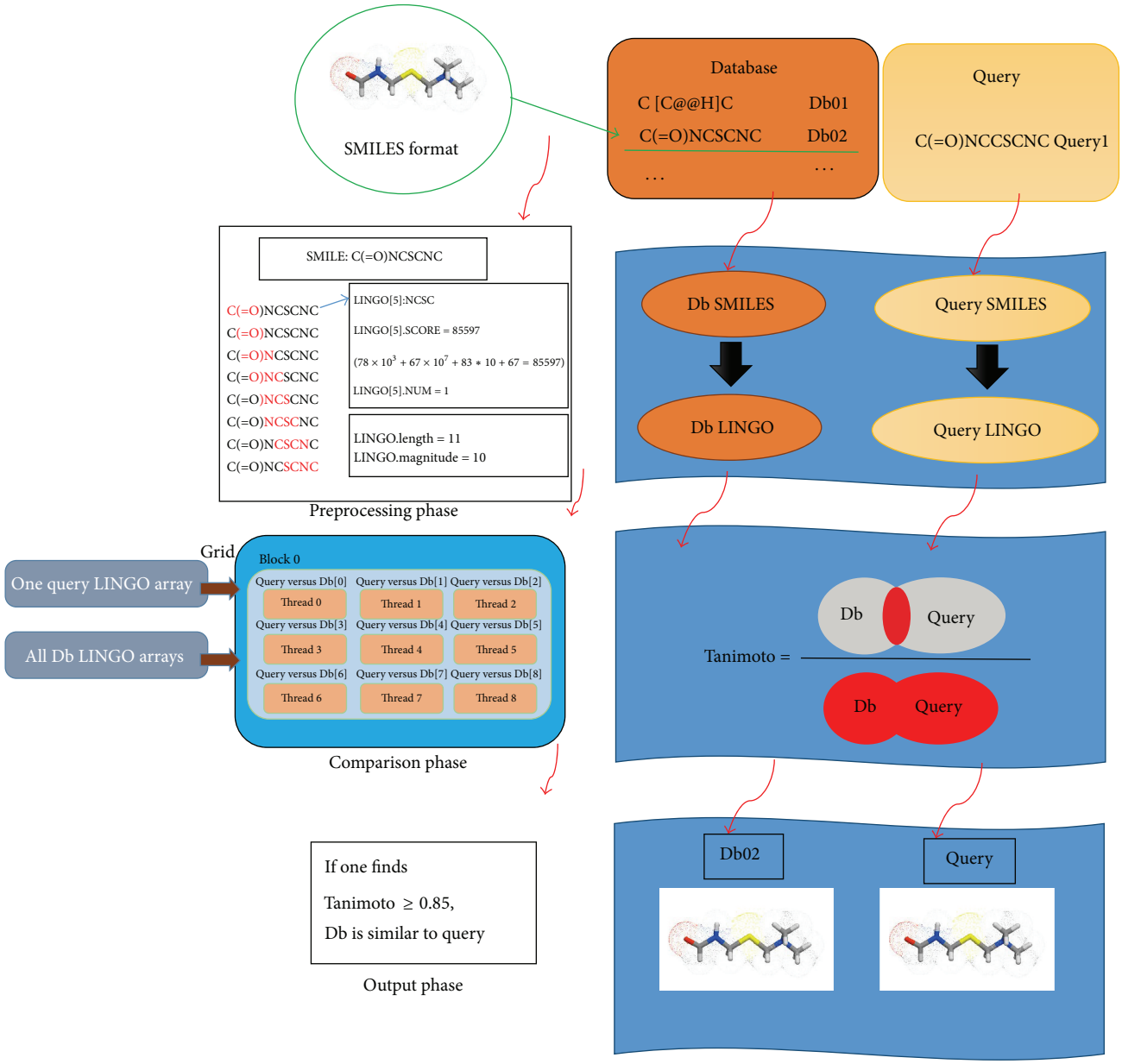


Figure 1: The flowchart of three phases in CUDA-MCC.

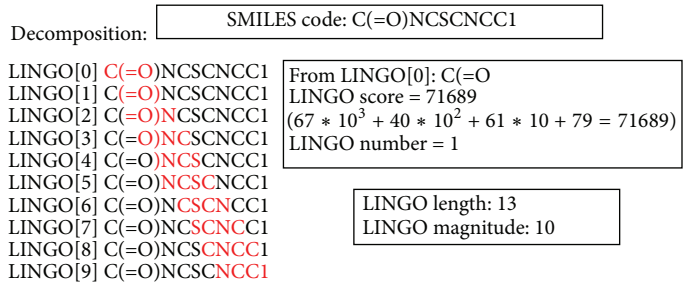


Figure 2: An example of a SMILES code in the *preprocessing phase*.

```
LINGO constructor
struct Lingo
{
   int* Score;
   int Length;
   int Magnitude;
   int* Number;
   char* index;
   int sumOfScore;
   int sumOfNum;
}
LINGO Number
for (int x = 0; x < Db_Type; x++)
    Db[x].Magnitude = 0;
    int len = Db[x].Length;
    for (int y = 0; y < len; y++)
        for (int z = 0; z < len; z++)
            if (Db[x].Score[y] == Db[x].Score[z])
                Db[x].Number[y]++;
LINGO Score
    for (int x = 0; x < Db_Type; x++)
        int len = Db[x].Length;
        for (int y = 0; y < len; y++)
            for (int z = 0; z < 4; z++)
                Db[x].Score[y]+ = (int)(((int)Db_smile[x][y + z] + 128) * pow (10, z));
LINGO Magnitude
for (int x = 0, tmp = 0; x < Db_Type; x++)
    tmp+ = Db[x].Length;
    for (int y = 0; y < Db[x].Length; y++)
        Db[x].Magnitude = Db[x].Number[y];
LINGO Length
for (int x = 0; x < Db_Type; x++)
    int len = strlen(Db_smile[x]);
    Db[x].Length = len;
```

Pseudocode 1: Pseudocodes of LINGO constructor, LINGO number, LINGO score, LINGO magnitude, and LINGO length.

the LINGO numbers of each 4-Lingo; a one-dimensional integer array is used to store the LINGO lengths of each compound; a one-dimensional integer array is used to store the LINGO magnitudes of each compound. These arrays mentioned above can be packaged in a complex structure array (LINGO constructor). The information was stored in a complex structure array for *Query* and *Database*, respectively, as shown in Figure 1.

The pseudocodes of LINGO constructor, LINGO number, LINGO score, LINGO magnitude, and LINGO length are listed in Pseudocode 1.

In this paper, we focused on the *comparison phase* implemented on GPU. Therefore, the above three parts are all done on CPU in order to simplify the problem in CUDA-MCC. In practice, the second and third parts also can be implemented on GPUs. When the second part is implemented on GPU, the memory usage should be considered. Since the number of LINGO magnitudes is unknown after the *reading files* part, the fixed (large) memory space of structure array in each compound should be allocated on GPU at first. By this way, the memory usage may be large and most memory space is wasted. Four LINGO types of each compound can be calculated by a thread; hence, thousands of concurrent threads can be used to process these compounds quickly. After that, the wasted memory space could be removed by the memory reallocated procedure and the prefix sum algorithm. It is easy to reallocate the previous structure array into a new one on GPU by all threads according to the indices, which is computed by GPU-based prefix sum algorithm published in the past. For the implementation of the third part on GPU, many GPU-based sorting algorithms have been proposed in the past. These GPU-based sorting algorithms can be modified to sort these LINGO scores quickly on GPU. In the following experimental tests, the computation time of *preprocessing phase* is not included in the time analysis; it only includes the time of *comparison phase* and *output phase*.

3.2. Comparison Phase. Before designing the *comparison phase* in CUDA-MCC, how to assign the comparison tasks on GPUs should be discussed. To compare two sets of compounds (A2A) can be seen as to compare a compound with a set of compounds repeatedly (O2A). Therefore, in CUDA-MCC, a compound in *Query* will be used to compare

```
_global_void siml (const struct Lingo* Q_d, const struct Lingo* Db_d,
                   const int* Q_Pos, const int* Db_Pos,
                   const int* Q_Score, const int* Q_Number,
                   const int* Db_Score, int* Db_Number,
                   float* Tanimoto, int x)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  float sum = 0;
  int insct = 0;
  int i = 0;
  int j = 0;
  float t = 0;
  while (i < Q_d[x].Length && j < Db_d[idx].Length)
    if (Q_Score[i + Q_Pos[x]] == Db_Score[Db_Pos[idx] + j])
      if (Q_Number[i + Q_Pos[x]] < Db_Number[Db_Pos[idx] + j])
        insct+ = Q_Number[i + Q_Pos[x]];                    i++; j++;
      else
        insct+ = Db_Number[Db_Pos[idx] + j];                i++; j++;
    else
      if (Q_Score[i + Q_Pos[x]] < Db_Score[Db_Pos[idx] + j])  i++;
      else                                                    j++;
  sum = Q_d[x].Magnitude + Db_d[idx].Magnitude;
  t = (insct/(sum − insct));
  if (t > 0.85)
  Tanimoto[x] = t;
}
```

Pseudocode 2: Pseudocode of computing Tanimoto coefficient.

with all compounds in *Database* by all threads on GPUs when executing the kernel function once. In a thread block, each thread is used to compare a compound in *Query* with one of the compounds in *Database*. Hence, the computing workload of each thread in a thread block should be equal in order to achieve the high performance. For multi-GPUs, the computing workload of each GPU card should also be the same.

A load-balancing strategy can be used to accelerate the computation speed for single- and multi-GPU cards. However, how to estimate the computing workload of a comparison is a problem, since only a compound in *Query* is used when executing the kernel function once. The computing workloads only need to consider the compounds in *Database*. The LINGO length and LINGO magnitude can be used directly as the measurements of computing workloads. The sums of LINGO scores and LINGO numbers can be calculated as shown in the pseudocodes of *preprocessing phase*, respectively, and then they are used as other measurements of computing workloads. Four LINGO-based load-balancing strategies are applied into CUDA-MCC by considering these four LINGO types. As mentioned in the *preprocessing phase*, the list of compounds in *Database* (structure array of *Database*) is sorted on CPU. In the *comparison phase*, on single-GPU, the structure arrays of *Query* (*Q*) and *Database* (*Db*) are transferred from CPU to GPU by using the following two libraries:

cudaMemcpy(Db_d, Db, sizeof(struct Lingo)∗Db_Type, cudaMemcpyHostToDevice);

cudaMemcpy(Q_d, Q, sizeof(struct Lingo)∗Q_Type, cudaMemcpyHostToDevice).

These two structure arrays are stored in the global memory. The time of transferring structure arrays from CPU to GPU is included in the time of *comparison phase*. A two-dimensional floating point array (result array) with a size of $k \times r$, where k and r are the number of compounds in *Query* and *Database*, respectively, is created in the global memory. This array is used to store the computed Tanimoto coefficient for each comparison. For each comparison, the sorted LINGO scores of a compound in *Query* are used to compare with other sorted LINGO scores of a compound in *Database*. It is a simple job to find the same LINGO scores among these two compounds. By accumulating the LINGO numbers corresponding to the same LINGO scores, the Tanimoto coefficient of a comparison can be computed.

The pseudocode of computing Tanimoto coefficient is listed in Pseudocode 2.

On multi-GPUs, the structure array of *Query* will be divided into several parts according to the computing capabilities of GPU cards. Then, these structure subarrays of *Query* are transferred from CPU to the global memory of the corresponding GPU, respectively. The complete structure array of *Database* also is transferred from CPU to the global

```
If (Tanimoto[idx + threadShift] > 0.85)
device_Tanimoto[idx] = Tanimoto[idx + threadShift];
cudaMemcpy(Tanimoto, device_Tanimoto, sizeof(int)* D_Type *Query_Type, cudaMemcpyDeviceToHost);
```

Pseudocode 3: Pseudocode of output phase.

memory of each GPU, respectively. For each GPU, a two-dimensional floating point array (partial result array) with a size of $m \times r$, where m is the number of compounds in *Query* assigned to this GPU, is created in the global memory to store the computed Tanimoto coefficient for each comparison. The job of each comparison is similar to that on a single-GPU.

3.3. Output Phase. After the *comparison phase*, on single-GPU, the result array is transferred from GPU to CPU; on multi-GPUs, the partial result arrays are transferred from GPUs to CPU, respectively, and then these partial result arrays are merged into a complete result array. For each compound in *Query*, the compounds in *Database* with more than 0.85 Tanimoto coefficients are reported on CPU, respectively. The threshold 0.85 in CUDA-MCC is set up in order to report the compounds with the most (possible) similar structure. It can be adjusted to report the results according to the requirements, even for complete results.

The pseudocode of *output phase* is listed in Pseudocode 3.

4. Experiment Results

In this work, CUDA-MCC was implemented by C+OpenMP+CUDA. In order to evaluate CUDA-MCC on single- and multi-GPUs, two machines are used in the experimental tests. The first machine has eight CPU cores; each core is Intel Xeon E5-2670 CPU of 2.6 GHz and single NVIDIA Tesla K20m GPU card with 2496 core of each 0.71 GHz. The second machine has eight CPU cores; each core is Intel Xeon E5-2650 CPU of 2.0 GHz and dual-NVIDIA Tesla K20m GPU card. There are three test sets used in the following tests: (s1) ten thousand compounds in *Query* and *Database*, respectively, (s2) thirty thousand compounds in *Query* and *Database*, respectively, and (s3) fifty thousand compounds in *Query* and *Database*, respectively. The test compounds are randomly selected from the ZINC database.

The first test is to evaluate CUDA-MCC for these three test sets on single NVIDIA Tesla K20m GPU card in machines 1 and 2. Since the number of threads in a thread block will affect the performance by CUDA-MCC, various numbers of threads in a thread block are used in the tests. In addition, four LINGO-based load-balancing strategies are also applied into these three test sets by considering the LINGO score (denoted by S), LINGO number (denoted by N), LINGO length (denoted by L), and LINGO magnitude (denoted by M). Figure 3(a) shows the speedup ratios of various numbers of threads in a thread block for these three test sets by CUDA-MCC on single NVIDIA Tesla K20m GPU card in machine 1. The speedup ratios by CUDA-MCC for four LINGO-based load-balancing strategies in these three test sets on single NVIDIA Tesla K20m GPU card in machine 1 are shown in Figure 3(b). In Figure 3(a), the speedup ratio increases when the number of threads in a thread block increases. Overall, 1024 threads in a thread block have the best speedup ratios among three test sets, and CUDA-MCC achieves 45 times faster than its CPU version on single NVIDIA Tesla K20m GPU card in machine 1. In Figure 3(b), CUDA-MCC achieves more than 35 times faster than its CPU version on single NVIDIA Tesla K20m GPU card in machine 1 for three test sets by each load-balancing strategy. The load-balancing strategies, S and M, outperform strategies L and N; however, the difference between them is very small. The performance and observations by CUDA-MCC on single NVIDIA Tesla K20m GPU card in machine 2 are similar to those in machine 1.

The second test is to evaluate CUDA-MCC for these three test sets on dual-NVIDIA Tesla K20m GPU card in machine 2. The goals of this test are to demonstrate that CUDA-MCC is useful for multi-GPUs, and the speedup ratios can be improved by using multi-GPU cards. Figure 4 shows the comparisons of speedup ratios by CUDA-MCC on single NVIDIA Tesla K20m GPU card and dual-NVIDIA Tesla K20m GPU card in machine 2 for three test sets. From Figure 4, the speedup ratios by CUDA-MCC on dual-NVIDIA Tesla K20m GPU card in machine 2 are significantly larger than those by CUDA-MCC on single NVIDIA Tesla K20m GPU card in machine 2. CUDA-MCC achieves 391 times faster than its CPU version on dual-NVIDIA Tesla K20m GPU card in machine 2. The execution time of CUDA-MCC includes ($t1$) the time of transferring structure arrays (structure subarrays on multi-GPUs) from CPU to (corresponding) GPU in the *comparison phase*, ($t2$) the time of comparing two sets of compounds in the *comparison phase*, ($t3$) the time of transferring (partial result arrays on multi-GPUs) result array from (corresponding) GPU to CPU in the *output phase*, and ($t4$) the reporting (and the time to merge partial result arrays into a complete result array on multi-GPUs) time to report the compounds with similar structure. When running CUDA-MCC on multi-GPUs, both sizes of structure array in *Query* and result array, needed to be transferred between CPU and one GPU, reduce. Therefore, times $t1$ and $t3$ reduce greatly and the speedup ratio increases.

5. Conclusion

In this paper, a GPU-based algorithm, CUDA-MCC, was proposed and implemented to do the MCC problem on single- and multi-GPUs. Four LINGO-based load-balancing strategies were applied into CUDA-MCC, and then discuss the effects by considering the LINGO score, LINGO number,

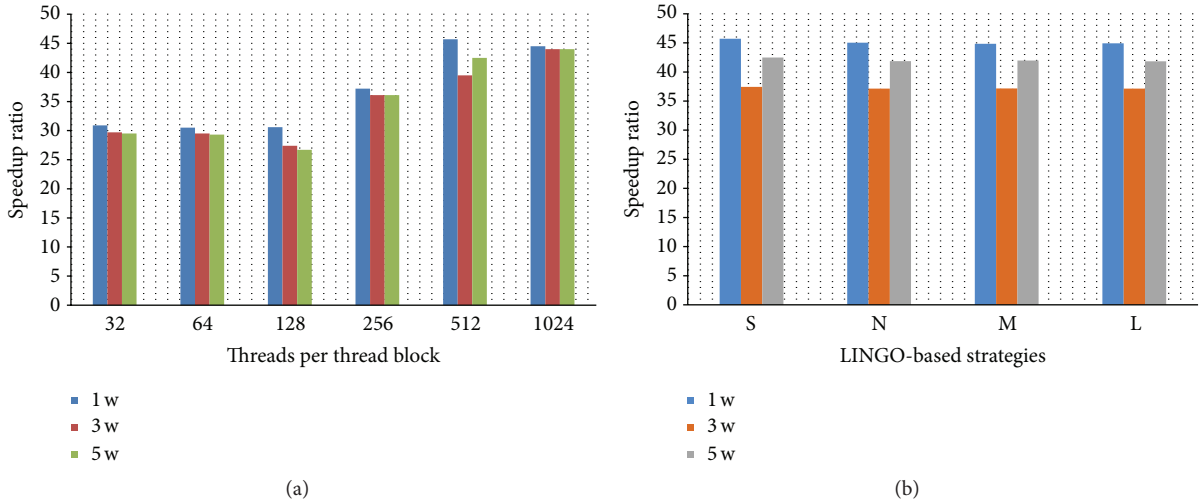


Figure 3: The speedup ratios for various numbers of threads in a thread block and four LINGO-based load-balancing strategies on three test sets and single NVIDIA Tesla K20m GPU card in machine 1.

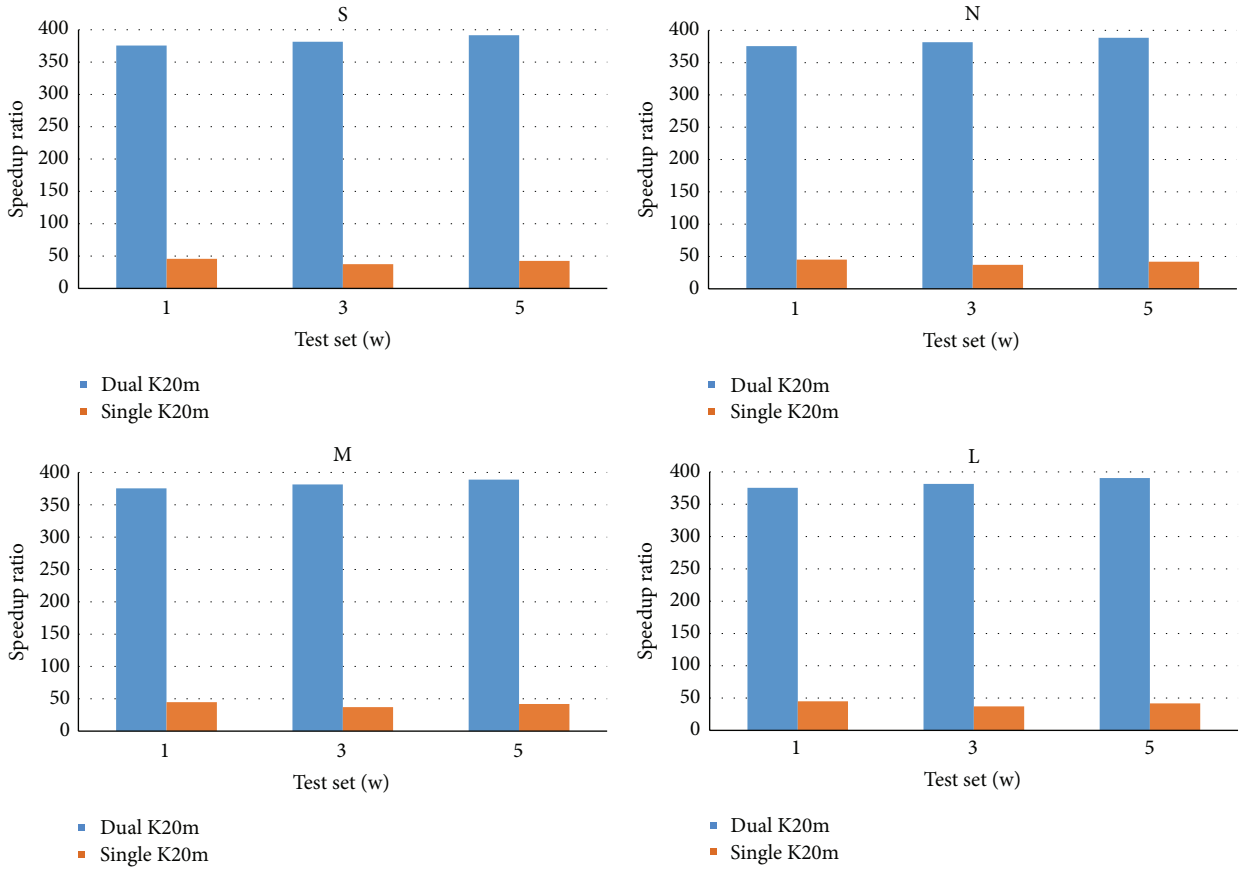


Figure 4: The comparisons of speedup ratios by CUDA-MCC on single NVIDIA Tesla K20m GPU card and dual-NVIDIA Tesla K20m GPU card in machine 2 for three test sets.

LINGO length, and LINGO magnitude, respectively. Two machines with single NVIDIA Tesla K20m GPU card and dual-NVIDIA Tesla K20m GPU card were used to evaluate CUDA-MCC, respectively. From experimental results, CUDA-MCC achieved 45 times and 391 times faster than its CPU version on single NVIDIA Tesla K20m GPU card and dual-NVIDIA Tesla K20m GPU card, respectively. Two observations were summarized in this work: (1) the speedup ratio increases when the number of threads in a thread block increases and (2) the LINGO score and LINGO magnitude

strategies outperform other two strategies. However, the difference between them is very small. There are two possible reasons. First, the size of test sets is not enough to show the difference. Second, the benefits of estimating the computing workload by using these four LINGO types are the same. CUDA-MCC is useful for O2A and A2A comparisons on single- and multi-GPUs.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

Part of this work was supported by the Ministry of Science and Technology under Grants MOST 104-2221-E-182-050, MOST 104-2221-E-182-051, MOST 103-2221-E-126-013, and MOST 103-2632-E-126-001-MY3. The authors would like to thank the anonymous reviewers and experts who discussed the paper with them.

References

[1] H. V. D. John, "Computer-aided drug design: the next 20 years," *Journal of Computer-Aided Molecular Design*, vol. 21, no. 10-11, pp. 591–601, 2007.

[2] A. R. Leach, B. K. Shoichet, and C. E. Peishoff, "Prediction of protein-ligand interactions. Docking and scoring: successes and gaps," *Journal of Medicinal Chemistry*, vol. 49, no. 20, pp. 5851–5855, 2006.

[3] M. Ravikumar, S. Pavan, S. Bairy et al., "Virtual screening of cathepsin K inhibitors using docking and pharmacophore models," *Chemical Biology and Drug Design*, vol. 72, no. 1, pp. 79–90, 2008.

[4] W. F. DeGrado, C. M. Summa, V. Pavone, F. Nastri, and A. Lombardi, "De novo design and structural characterization of proteins and metalloproteins," *Annual Review of Biochemistry*, vol. 68, pp. 779–819, 1999.

[5] R. R. S. Pissurlenkar, M. S. Shaikh, and E. C. Coutinho, "3D-QSAR studies of Dipeptidyl peptidase IV inhibitors using a docking based alignment," *Journal of Molecular Modeling*, vol. 13, no. 10, pp. 1047–1071, 2007.

[6] A. Lauria, M. Ippolito, M. Fazzari et al., "IKK-beta inhibitors: an analysis of drug-receptor interaction by using molecular docking and pharmacophore 3D-QSAR approaches," *Journal of Molecular Graphics and Modelling*, vol. 29, no. 1, pp. 72–81, 2010.

[7] J. J. Irwin and B. K. Shoichet, "ZINC-A free database of commercially available compounds for virtual screening," *Journal of Chemical Information and Modeling*, vol. 45, no. 1, pp. 177–182, 2005.

[8] Y. Wang, J. Xiao, T. O. Suzek, J. Zhang, J. Wang, and S. H. Bryant, "PubChem: a public information system for analyzing bioactivities of small molecules," *Nucleic Acids Research*, vol. 37, no. 2, pp. W623–W633, 2009.

[9] L. C. Blum and J.-L. Reymond, "970 Million druglike small molecules for virtual screening in the chemical universe database GDB-13," *Journal of the American Chemical Society*, vol. 131, no. 25, pp. 8732–8733, 2009.

[10] N. Nikolova and J. Jaworska, "Approaches to Measure Chemical Similarity—a Review," *QSAR and Combinatorial Science*, vol. 22, no. 9-10, pp. 1006–1026, 2004.

[11] J. Bajorath, "Integration of virtual and high-throughput screening," *Nature Reviews Drug Discovery*, vol. 1, no. 11, pp. 882–894, 2002.

[12] Y. C. Martin, J. L. Kofron, and L. M. Traphagen, "Do structurally similar molecules have similar biological activity?" *Journal of Medicinal Chemistry*, vol. 45, no. 19, pp. 4350–4358, 2002.

[13] D. Vidal, M. Thormann, and M. Pons, "LINGO, an efficient holographic text based method to calculate biophysical properties and intermolecular similarities," *Journal of Chemical Information and Modeling*, vol. 45, no. 2, pp. 386–393, 2005.

[14] J. A. Grant, J. A. Haigh, B. T. Pickup, A. Nicholls, and R. A. Sayle, "Lingos, finite state machines, and fast similarity searching," *Journal of Chemical Information and Modeling*, vol. 46, no. 5, pp. 1912–1918, 2006.

[15] T. Oliver, B. Schmidt, D. Nathan, R. Clemens, and D. L. Maskell, "Using reconfigurable hardware to accelerate multiple sequence alignment with ClustalW," *Bioinformatics*, vol. 21, no. 16, pp. 3431–3432, 2005.

[16] T. F. Oliver, B. Schmidt, and D. L. Maskell, "Reconfigurable architectures for bio-sequence database scanning on FPGAs," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 52, no. 12, pp. 851–855, 2005.

[17] I. T. S. Li, W. Shum, and K. Truong, "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)," *BMC Bioinformatics*, vol. 8, article 185, 2007.

[18] A. Szalkowski, C. Ledergerber, P. Krähenbühl, and C. Dessimoz, "SWPS3—fast multi-threaded vectorized Smith-Waterman for IBM Cell/B.E. and ×86/SSE2," *BMC Research Notes*, vol. 1, article 107, 2008.

[19] M. S. Farrar, "Optimizing Smith-Waterman for the Cell Broad Band Engine," http://cudasw.sourceforge.net/sw-cellbe.pdf.

[20] A. Wirawan, C. K. Kwoh, N. T. Hieu, and B. Schmidt, "CBESW: sequence alignment on the playstation 3," *BMC Bioinformatics*, vol. 9, article 377, 2008.

[21] Y. Liu, W. Huang, J. Johnson, and S. H. Vaidya, "GPU accelerated Smith-waterman," in *Computational Science—ICCS 2006*, vol. 3994 of *Lecture Notes in Computer Science*, part 4, pp. 188–195, Springer, Berlin, Germany, 2006.

[22] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, IEEE, Rhodes Island, Greece, April 2006.

[23] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *ACM Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[24] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, pp. 948–960, 1972.

[25] I. S. Haque, V. S. Pande, and W. P. Walters, "SIML: a fast SIMD algorithm for calculating LINGO chemical similarities on GPUs and CPUs," *Journal of Chemical Information and Modeling*, vol. 50, no. 4, pp. 560–564, 2010.

[26] C. Ma, L. Wang, and X.-Q. Xie, "GPU accelerated chemical similarity calculation for compound library comparison," *Journal of Chemical Information and Modeling*, vol. 51, no. 7, pp. 1521–1527, 2011.

[27] Q. Liao, J. Wang, Y. Webster, and I. A. Watson, "GPU accelerated support vector machines for mining high-throughput screening data," *Journal of Chemical Information and Modeling*, vol. 49, no. 12, pp. 2718–2725, 2009.