INT. J. BIOAUTOMATION, 2019, 23(1), 117-129 doi: 10.7546/ijba.2019.23.1.117-129

An Improved Smith-Waterman Algorithm Based on Spark Parallelization

Yanfeng Liu[1], Leixiao Li[1,2,3*], Jing Gao[2]

[1]*College of Data Science and Application*
Inner Mongolia University of Technology
Hohhot 010080, China
E-mail: 2424568699@qq.com

[2]*College of Computer and Information Engineering*
Inner Mongolia Agricultural University
Hohhot 010018, China
E-mails: llxhappy@126.com, gaojing@imau.edu.cn

[3]*Inner Mongolia Autonomous Region Engineering &*
Technology Research Center of Big Data Based Software Service
Inner Mongolia University of Technology
Hohhot 010080, China

[*]*Corresponding author*

Received: April 10, 2018 **Accepted: March 05, 2019**

Published: March 31, 2019

Abstract: *This paper proposes the design and the implementation of a Spark parallelization plan for improving the Smith-Waterman (SW) algorithm, named the Spark-OSW algorithm. Then, the Spark-OSW was verified through accuracy, performance and acceleration tests. The results show that the proposed algorithm achieved 100% accuracy, ran much faster than the SW, and performed well in cluster environment. The research findings shed important new light on the database search for gene sequences.*

Keywords: *Sequence alignment, Smith-Waterman algorithm, Spark parallelization, Spark-OSW algorithm.*

Introduction

As an essential operation in bioinformatics, sequence alignment has immense popularity in many fields, ranging from disease diagnosis, drug engineering to biomaterial engineering [1]. Smith-Waterman (SW) algorithm is a commonly used approach of sequence alignment, thanks to its high accuracy. Nevertheless, this pairwise sequence alignment algorithm is too complex in space and time to achieve desirable computing results. This defect is magnified with the rapid growth of reference databases and long sequence alignment.

Many techniques have been implemented to optimize the SW algorithm, such as graphic processing unit (GPU) [5, 7, 8, 11-13], message passing interface (MPI) [17], cluster [3, 4, 9], single instruction multiple data (SIMD) [2, 16, 18], field programmable gate array (FPGA) [14], etc. [20]. However, none of these optimized SW algorithms can fully satisfy the sequence alignment demand in the big data era, owing to the defect in algorithm parallelization.

With the development of big data technology, more and more scholars are now using Hadoop [15] and Spark technology to achieve algorithm parallelization. Despite its advantages in computing speed and big data processing, the Spark technology, with its root in memory computing, has not been widely adopted by biologists for parallelizing sequence alignment algorithms.

Zhao et al. [18] were the first to implement the SW algorithm in the distributed computing framework on Apache Spark. The implementation managed to improve the system speed and calculate the similar sequences, but failed to achieve a high applicability or optimize the local alignment.

Xu et al. [16] relied on the Spark to enable the SW-based SIMD in a horizontally scalable distributed environment, and successfully calculated the most similar sequences; however, this attempt could not solve the exact local optimal alignment, and the SIMD command set is featured by high complexity and low code portability.

In light of the above, this paper proposes the design and the implementation of a Spark parallelization plan for improving the SW algorithm, named the Spark-OSW algorithm. Then, the Spark-OSW was verified through experiments. The results show that this algorithm enjoys low time complexity, fast speed, light computing load and good performance.

The paper is organized as follows: Section 2 presents a review the literature on the SW and the Spark. Section 3 presents an improvement of the SW algorithm. In Section 4 the parallelization of SW algorithm on Spark platform is described and the Spark-OSW algorithm is introduced. Section 5 presents some tests of the accuracy of Spark-OSW algorithm and discusses the experimental results. In Section 6 several conclusions are done.

Literature review

Smith-Waterman algorithm

The SW algorithm is a dynamic programming strategy looking for the local optimal alignment between two gene (protein) sequences. Originating from the Needleman-Wunsch algorithm, the SW algorithm mainly performs double sequence alignment in the local range. This algorithm can outperform others in the accuracy of double and multiple sequence alignments at the price of extremely high complexity in space and time.

Let *s* and *t* be two gene sequences (Fig. 1), whose lengths are m and n, respectively. Then, the *i*-th character of sequence *s* can be denoted as s_i, $(1 \le i \le m)$ and the *j*-th character of sequence t can be denoted as t_j, $(1 \le j \le n)$.

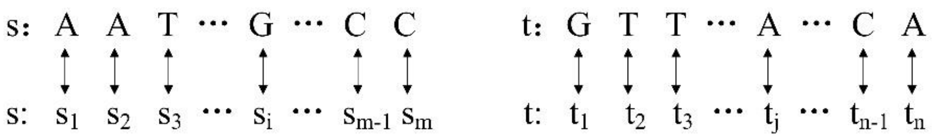

Fig. 1 Gene sequences *s* and *t*

Let *D* be the score matrix (Fig. 2) of the two sequences, and d_{ij} be the element in the *i*-th row and *j*-th column of the score matrix.

	Seq t:	t_1	t_2	···	t_j	···	t_{n-1}	t_n
		d_{00}	d_{01}	d_{02}	···	d_{0j}	···	$d_{0,n-1}$	d_{0n}
Seq s:	s_1	d_{10}	d_{11}	d_{12}	···	d_{1j}	···	$d_{1,n-1}$	d_{1n}
	s_2	d_{20}	d_{21}	d_{22}	···	d_{2j}	···	$d_{2,n-1}$	d_{2n}
	···	···	···	···	···	···	···	···	···
	s_i	d_{i0}	d_{i1}	d_{i2}	···	d_{ij}	···	$d_{i,n-1}$	d_{in}
	···	···	···	···	···	···	···	···	···
	s_{m-1}	$d_{m-1,0}$	$d_{m-1,1}$	$d_{m-1,2}$	···	$d_{m-1,j}$	···	$d_{m-1,n-1}$	$d_{m-1,n}$
	s_m	d_{m0}	d_{m1}	d_{m2}	···	d_{mj}	···	$d_{m,n-1}$	d_{mn}

Fig. 2 Score matrix D

The SW algorithm is implemented in the following steps:

(1) Initialization: Assign zero to the element in the first row and first column of score matrix *D*, that is, $d_{0j} = 0, (1 \le j \le n)$ and $d_{i0} = 0, (1 \le i \le m)$ (Fig. 3)

	Seq t:	t_1	t_2	···	t_j	···	t_{n-1}	t_n	
	$d_{00}=0$	$d_{01}=0$	$d_{02}=0$	···	$d_{0j}=0$	···	$d_{0,n-1}=0$	$d_{0n}=0$	first row
Seq s: s_1	$d_{10}=0$								
s_2	$d_{20}=0$								
···	···								
s_i	$d_{i0}=0$								
···	···								
s_{m-1}	$d_{m-1,0}=0$								
s_m	$d_{m0}=0$								
	first column								

Fig. 3 Initialization of the element as zero in the first row and first column

(2) Calculation of score matrix *D*: The score function of score matrix *D* can be expressed as:

$$\begin{cases} p(a,a)=1 \\ p(a,b)=0,\ (a \neq b) \\ p(a,\text{-})=p(\text{-},b)=\text{-}1 \end{cases}, \tag{1}$$

where $p(a,a)=1$ means the score is 1 when the two characters match; $p(a,b)=0, (a \neq b)$ means the score is 0 when the two characters do not match; $p(a,\text{-}) = p(b,\text{-}) = \text{-}1$ means the score is -1 when one of the two characters is vacant.

The element d_{ij} in score matrix *D* can be calculated as:

$$d_{ij} = \max\begin{cases} d_{i\text{-}1,j} + p(s_i,\text{-}) \\ d_{i,j\text{-}1} + p(\text{-},t_j) \\ d_{i\text{-}1,j\text{-}1} + p(s_i,t_j) \\ 0 \end{cases}, \ (1 \le i \le m,\ 1 \le j \le n). \tag{2}$$

Calculate the value of each element d_{ij} in the score matrix according to Eqs. (1) and (2) to form the score matrix *D*.

(3) Backtracking: Find the largest element d_{ij} in score matrix *D*, determine whether d_{ij} is calculated from $d_{i,j\text{-}1}$, $d_{i\text{-}1,j}$ or $d_{i\text{-}1,j\text{-}1}$, and write down the result. Repeat this process until reaching an element whose value is zero. In this way, a backtracking path can be obtained (Fig. 4).

	Seq t:	···	t_q	···	t_{j-1}	t_j	···	t_n
	d_{00}	···	d_{0q}	···	$d_{0,j-1}$	d_{0j}	···	d_{0n}
Seq s: ···	···	···	···	···	···	···	···	···
s_p	d_{p0}	···	$\mathbf{d_{pq}=0}$	···	$d_{p,j-1}$	d_{pj}	···	d_{pn}
s_{p+1}	$d_{p+1,0}$	···	$\mathbf{d_{p+1,q}}$	···	$d_{p+1,j-1}$	$d_{p+1,j}$	···	$d_{p+1,n}$
···	···	···	···	**···**	···	···	···	···
s_{i-2}	$d_{i-2,0}$	···	$d_{i-2,q}$	···	$\mathbf{d_{i-2,j-1}}$	$d_{i-2,j}$	···	$d_{i-2,n}$
s_{i-1}	$d_{i-1,0}$	···	$d_{i-1,q}$	···	$\mathbf{d_{i-1,j-1}}$	$d_{i-1,j}$	···	$d_{i-1,n}$
s_i	d_{i0}	···	$d_{i,q}$	···	$d_{i,j-1}$	$\mathbf{d_{ij}}$	···	d_{in}
···	···	···	···	···	···	···	···	···
s_m	d_{m0}	···	$d_{m,q}$	···	$d_{m,j-1}$	$d_{m,j}$	···	d_{mn}

Fig. 4 Backtracking path

(4) Solving local optimal alignment: Starting with the largest element d_{ij} of score matrix *D*, search for the local optimal alignment through reverse backtracking according to the path generated in Step 3 until reaching an element whose value is zero. During the backtracking process, if d_{ij} comes from $d_{i\text{-}1,j}$, compare s_i with "-"; if d_{ij} comes from $d_{i,j\text{-}1}$, compare "-" with t_j; if d_{ij} comes from $d_{i\text{-}1,j\text{-}1}$, compare s_i with t_j.

The Spark

The Spark is a fast, versatile and highly open cluster computing platform [19]. The memory-based feature of Spark increases the speed of some applications by 100 times [6]. The design of this platform accommodates the functions of many other distributed systems, including batch processing, iterative computing, inter-query (Hive) and stream processing (Strom). Moreover, the Spark provides various interfaces in addition to the common application programming interfaces (APIs) for Python, Java, Scala, and Structured Query Language (SQL). The Spark integrates well with Hadoop, Kafka and other big data tools [6]. With the adoption of MapReduce, this platform boasts all the advantages of Hadoop, a typical distributed computing framework. More importantly, Spark is memory-based, that is, the intermediate computing result is stored in the memory, eliminating the need to transfer the result to hard drive.

This feature reduces the time consumed in the input and output processes. Therefore, the Spark outperforms Hadoop in the quantity and quality of big data processing.

The modules of the Spark are shown in Fig. 5.

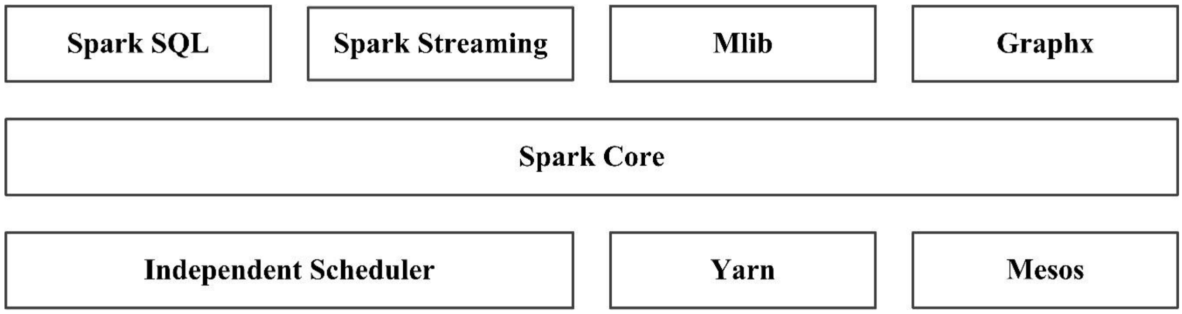

Fig. 5 The modules of the Spark [6]

As shown in Fig. 5, the Spark mainly consists of the Spark Score, Spark SQL, Spark Streaming, Mlib, Graphx and Cluster Manager. Specifically, the Spark Score contains the basic function of the Spark and defines the API for Spark programming. The main programming abstraction of Spark is Resilient Distributed Datasets (RDD) [10]. Similar to Hive SQL and MySQL, Spark SQL is a library for Spark to process structured databases. Spark Streaming is real-time data stream processing module. Similar to Strom, this module provides an API to process real-time streaming data. The Mlib is a package containing general machine learning features (e.g., classification, clustering and regression), model evaluation, data import and so on. The Graphx is a library that processes graphs and performs parallel graph calculations. Like Spark Streaming and Spark SQL, the Graphx inherits the RDD API, offers various graph operations and provides common graph algorithms (e.g., the Pang Rank algorithm). The Cluster Manager is a separate scheduler in the Spark for cluster management.

Improvement of the Smith-Waterman algorithm

In recent years, Li et al. [9] made an improvement to the SW algorithm by recording the source of element d_{ij} when its value is calculated, eliminating the need to compute its value in source tracking of Step. The improved algorithm has much less computing load than the original algorithm.

Inspired by Li's modification [9], this paper proposes the Opti-SW algorithm to reduce the load and complexity of the computation. Specifically, the first two rows and columns of the score matrix were initialized simultaneously and the calculation of the score matrix formula was simplified. The workflow of the Opti-SW algorithm is shown in Fig. 6.

Design of Spark-OSW algorithm

In light of the features of the Spark platform and the steps of the Opti-SW, the author prepared a plan to parallelize the Opti-SW algorithm on the Spark (Spark-OSW). Following this plan, the Spark-OSW algorithm calculates the score matrix, sorts the scores, selects the score matrix and solves the optimal alignment. The computing load is positively correlated with the data size, because the local optimal alignment is solved with only a part of the score matrix.

Workflow of Spark-OSW

The workflow of the Spark-OSW is illustrated in Fig. 7.

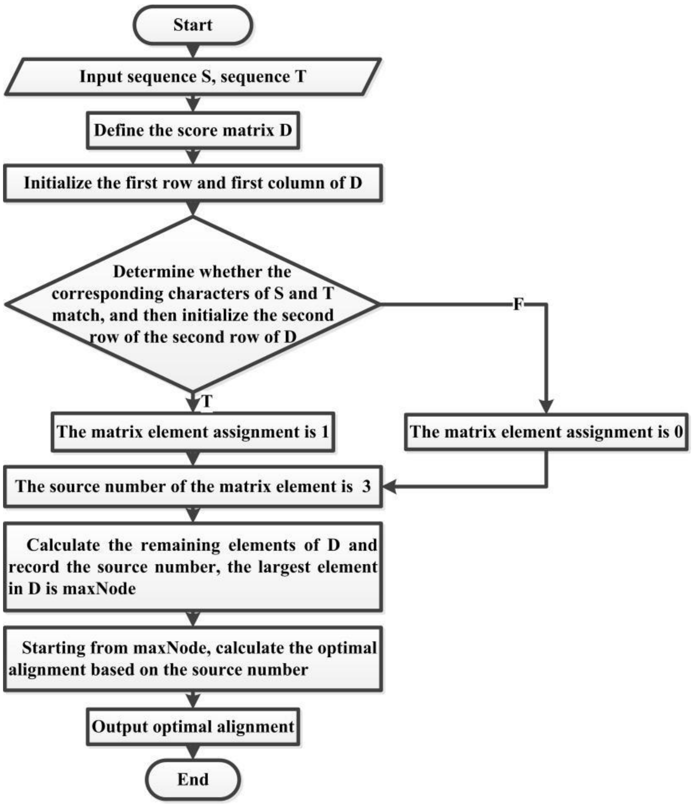

Fig. 6 Flowchart of Opti-SW algorithm

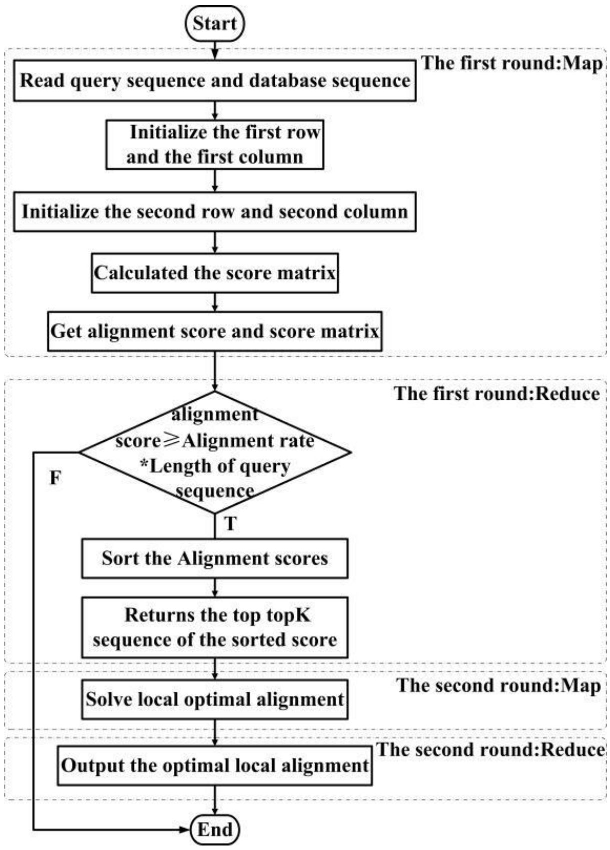

Fig. 7 Flowchart of Spark-OSW

Details of Spark-OSW

(1) The first round of Map task

Pair *RDD*1 = Pair *RDD*0.*map*()// *map*() function. The main task is illustrated in Steps 2-5.

Step 1: Spark's *textFile*() function reads the query sequence in Hadoop distributed file system (HDFS) to generate Pair *RDD*0. Complete Steps 2-5 after creating the query sequence and all sequence scores in the database.

Step 2: Initialize the first column and the first row. Initialize the first row and the first column elements of the score matrix, respectively, $d_{0j} = 0, (1 \le j \le n)$ and $d_{i0} = 0, (1 \le i \le m)$ as zero.

Step 3: Initialize the second row and second column. Initialize the second row and second column elements of the score matrix by improved equation and record the number of the element source.

> Note: The elements d_{1j} in the second row comes from $d_{0,j\text{-}1}, (1 < j \le n)$, while those d_{i1} in the second column comes from $d_{i\text{-}1,0}, (1 < i \le m)$. The $d_{i,j}$ from $d_{i\text{-}1,j}$, $d_{i,j\text{-}1}$ and $d_{i\text{-}1,j\text{-}1}$ are numbered as 1, 2 and 3, respectively. These numbers are always assigned as above in the following analysis.

Step 4: Calculate the score matrix. Calculate the score matrix *D* according to improved equation and record the number of the element source.

Step 5: Traverse the score matrix to obtain the alignment score *maxScore* and return Pair *RDD*1 <*maxScore*, *eachDbSeqName*>, where *MaxScore* is the value of the largest element in the matrix and *EachDbSeqName* is the name of the database sequence.

(2) The first round of Reduce task

Pair *RDD*2 = Pair *RDD*1.filter() // Pair *RDD*2 aims to return the sequence in Pair *RDD*1 that meets the score requirements.

Pair *RDD*2.*persist*()/*Use the *RDD.persist*() function to persist the alignment score of the Map stage.

> Note: The Spark automatically recalculates each RDD during the operation; the *RDD.persist*() function can be used to cache an RDD if it is favorable to reuse the RDD in multiple operations.

Pair *RDD*3 = Pair *RDD*2.*sortByKey*() // sort by alignment score.
Pair *RDD*4 = Pair *RDD*3.*take*(*topK*) // return the top *K* sequences with the highest score.

(3) The second round of Map task

Pair *RDD*5 = Pair *RDD*4.*map*() // main task of *map*(): solving the local optimal alignment.

Starting with the largest element d_{ij} of score matrix *D*, search for the local optimal alignment through reverse backtracking according to the path generated in Step 3 until reaching an element whose value is zero. During the backtracking process, if d_{ij} comes from $d_{i\text{-}1,j}$, compare s_i with

"-"; if d_{ij} comes from $d_{i,j-1}$, compare "-" with t_j; if d_{ij} comes from $d_{i-1,j-1}$, compare s_i with t_j. Pair *RDD*5 returns <*eachDbSeqName*, *OptimalAlignment*>, where *eachDbSeqName* is the sequence name and *OptimalAlignment* is the local optimal alignment.

(4) The second round of Reduce task

Pair *RDD*5.*foreach*(*println*) // *RDD.foreach*(): Output the comparison results.

Experimental verification

Accuracy tests

The Spark-OSW algorithm was compared with the SW algorithm through tests in the Spark platform.

The test data were extracted from the gene sequence database of the National Center for Biotechnology Information (NCBI: https://www.ncbi.nlm.nih.gov/). The input data of the Spark-OSW include: query sequence *queryFile* lengths of 2, 4, 8, 16, 32 and 64; database sequence *dbFile* length of 64, 32, 16, 8, 4 and 2; the number of fragments *splitNum* of 32; the number of tasks *taskNum* of 1; the number of top *k* outputs of 1; the comparison ratio (comparison score / query sequence length) * 100%) of identity is 0.0 (i.e., any sequence of comparison ratios may be outputted).

The input data of the SW algorithm include: query sequence *queryFile* lengths of 2, 4, 8, 16, 32 and 64; database sequence *dbFile* length of 64, 32, 16, 8, 4 and 2; the number of fragments *splitNum* of 32; the number of tasks *taskNum* of 1.

Table 1 compares the test results of the two algorithms for different query sequences in different database sequences.

As shown Table 1, the Spark-OSW algorithm and the SW algorithm both achieved the optimal local alignment under different sequences and comparison ratios, indicating that the Spark-OSW is also an accurate sequence alignment algorithm.

Performance tests

The Spark-OSW algorithm was further compared with the SW algorithm through performance tests on a single-node Spark platform.

Table 1. Accuracy test results of the two algorithms

Query sequence length	Target sequence length	Comparison ratio		Is the optimal local contrast of the SW and Spark-OSW output consistent?	Spark-OSW accuracy
		SW	Spark-OSW		
2	64	100%	100%	√	100%
4	32	75%	75%	√	100%
8	16	62.5%	62.5%	√	100%
16	8	50%	50%	√	100%
32	4	12.5%	12.5%	√	100%
64	2	3.125%	3.125%	√	100%

The test data were extracted from the gene sequence database of the National Center for Biotechnology Information (NCBI: https://www.ncbi.nlm.nih.gov/). The input data of the Spark-OSW include: query sequence *queryFile* lengths of 48; database sequence *dbFile* size of 10MB, 20MB, 40MB, 80MB and 160MB (txt. files); the number of fragments *splitNum* of 32; the number of tasks *taskNum* of 1; the comparison ratio (comparison score / query sequence length)*100%) of identity is 1.0 (i.e., any sequence of comparison ratios may be outputted).

The input data of the SW algorithm include: query sequence *queryFile* lengths of 48; database sequence *dbFile* size of 10MB, 20MB, 40MB, 80MB and 160MB (txt. files); the number of fragments *splitNum* of 32; the number of tasks *taskNum* of 1.

The test results are recorded in Fig. 8, where the mean rate (R) is calculated as (SW runtime-Spark-OSW runtime)/ Spark-OSW*100% and the size (S) is the size of the database sequence file.

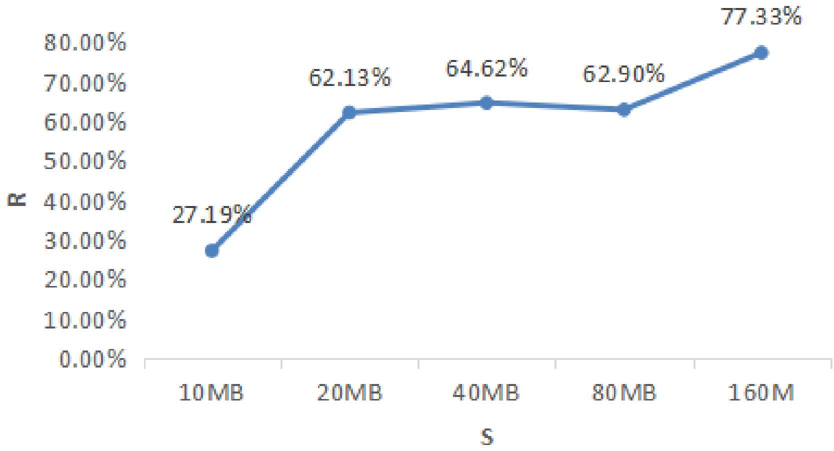

Fig. 8 Performance test results of the two algorithms

It can be seen from Fig. 8 that, the rate of the Spark-OSW is positively correlated with the data size within a certain range, i.e., the greater the data, the shorter the Spark-OSW runtime (the greater the SW runtime). The Spark-OSW is clearly more efficient than the contrastive algorithm. The high efficiency of the Spark-OSW is attributed to the optimized parallelization plan, which reduces the computing load proportionally to the data size.

Spark cluster tests

The test data were extracted from the gene sequence database of the National Center for Biotechnology Information (NCBI: https://www.ncbi.nlm.nih.gov/). A gene sequence of the length 48 was selected as the query sequence while a txt file of the size 20MB, 40MB, 60MB, 80MB, 100MB, 120MB, 140MB, 160MB, 180MB or 200MB was taken as the database sequence.

The Spark-OSW and the SW were operated ten times in a single-node Spark cluster, an eight-node Spark cluster, and a sixteen-node Spark cluster, respectively. The runtime of each algorithm under each condition is recorded as Figs. 9-11, where the Spark-OSW curve is the actual runtime of the Spark-OSW, the SW curve is the actual runtime of the OSW, the T-Spark-OSW curve is the theoretical runtime of the Spark-OSW and the T-SW is the theoretical runtime of the SW.

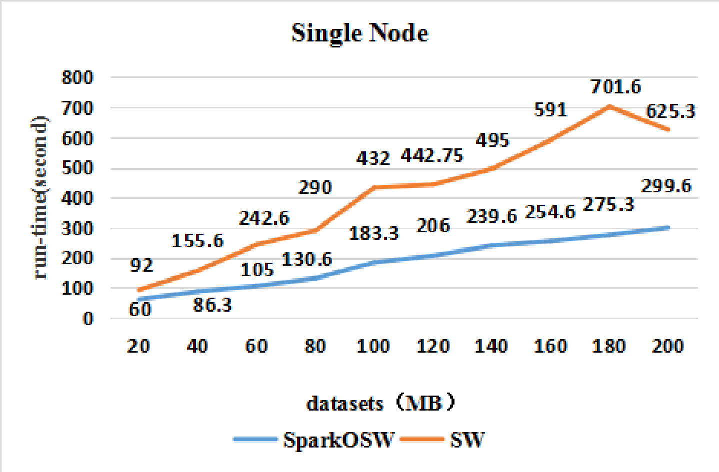

Fig. 9 Runtime of each algorithm in a single-node Spark cluster

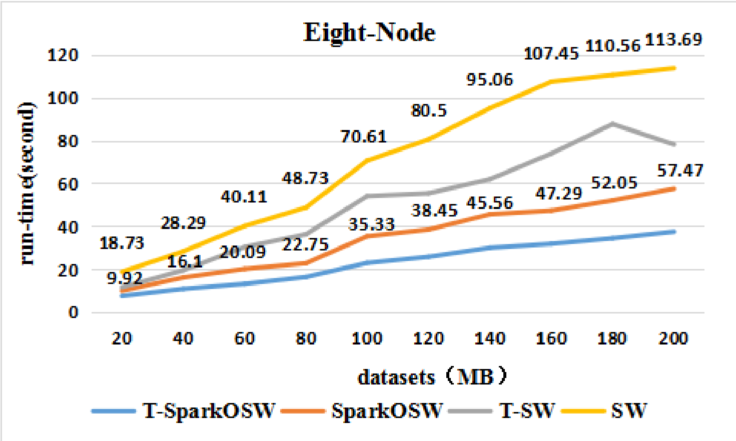

Fig. 10 Runtime of each algorithm in an eight-node Spark cluster

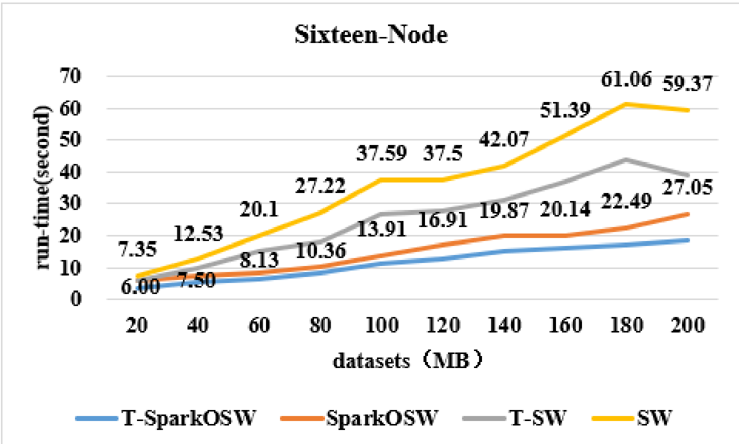

Fig. 11 Runtime of each algorithm in a sixteen-node Spark cluster

Then, the two algorithms each performed acceleration ratio experiments in the eight-node Spark cluster and the sixteen-node Spark cluster, respectively. The results of the two algorithms are compared in Fig. 12.

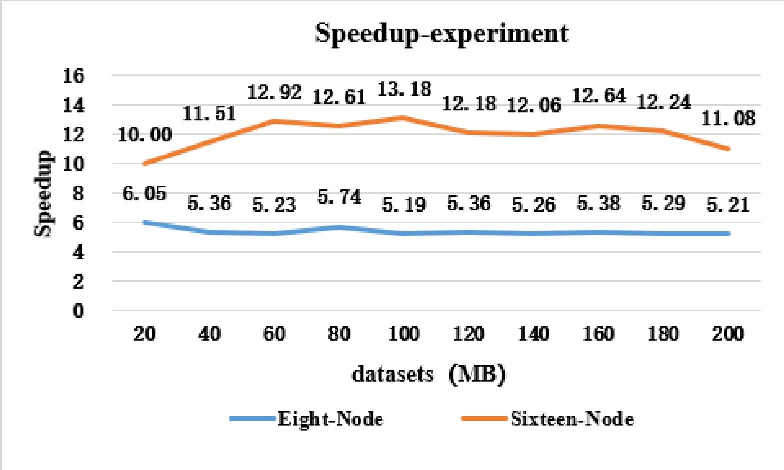

Fig. 12 Results of acceleration test

As shown in Fig. 12, the acceleration of the Spark-OSW remained in a certain range despite the growth of the data size. For this algorithm, the acceleration in the sixteen-node cluster was much faster than that in the eight-node cluster, revealing that the algorithm performance improves with the growth in the number of nodes. In theory, the acceleration should be proportional to the number of nodes. However, the acceleration did not reach eight in the eight-node cluster or sixteen in the sixteen-node cluster, an evidence to the transmission loss between the nodes.

Conclusions

This paper proposes the design and the implementation of a Spark parallelization plan for improving the SW algorithm, creating the Spark-OSW algorithm. Then, the Spark-OSW was verified through accuracy, performance and acceleration tests. The results show that the proposed algorithm achieved 100% accuracy, ran much faster than the SW, and performed well in cluster environment. In addition, the speed improvement, i.e., runtime reduction, is more obvious with the growth in data size.

Acknowledgement

The work is funded in part by the National Natural Science Foundation of China (NSFC) under Grant No. 61462070, the Doctoral research fund project of Inner Mongolia Agricultural University under Grant No. BJ09-44 and the Inner Mongolia Autonomous Region Key Laboratory of big data research and application for agriculture and animal husbandry.

References

1. Chen M. (2016). Instant Notes in Bioinformatics, Beijing, Science Press (in Chinese).
2. Daily J. (2016). Parasail: SIMD C Library for Global, Semi-global, and Local Pairwise Sequence Alignments, BMC Bioinformatics, 17(1), 1-6.
3. Feng B. L. (2015). The Research of Distributed Parallel Optimization of Pairwise Sequence Alignment Needleman-Wunsch Algorithm, Hohhot: Inner Mongolia Agricultural University (in Chinese).
4. Feng B., J. Gao (2016). Distributed Parallel Needleman-Wunsch Algorithm on Heterogeneous Cluster System, International Conference on Network & Information Systems for Computers, 358-361.
5. Gao J., Y. Jiao, W. G. Zhang (2014). Overview of Sequence Alignment for High-throughput Sequencing Data, Life Science Research, 18(5), 458-464 (in Chinese).

6. Holden K. (2015). Spark Fast Big Data Analysis, Beijing, People's Posts and Telecommunications Press.
7. Jain C., S. Kumar (2014). Fine-grained GPU Parallelization of Pairwise Local Sequence Alignment, International Conference on High Performance Computing, IEEE Computer Society, 1-10.
8. Li C., Y. L. Xu, C. B. Deng (2015). DNA Pairwise Sequence, Journal of Computer Systems, 24(9), 112-117.
9. Li D. W. (2011). Research on Parallel Algorithm of Sequence Alignment Based on Dynamic Programming, Journal of Jinggangshan University (Natural Science Edition), 128(1), 123-128. (in Chinese)
10. Liu J., Y. Liang, N. Ansari (2016). Spark-based Large-scale Matrix Inversion for Big Data Processing, IEEE Access, 4, 2166-2176.
11. Liu Y., A. Wirawan, B. Schmidt (2013). CUDASW++3.0: Accelerating Smith-Waterman Protein Database Search by Coupling CPU and GPU SIMD Instructions, BMC Bioinformatics, 14, 117-121.
12. Liu Y., X. L. Wang, J. Y. Li, Y. Q. Mao, D. S. Zhao (2012). Advances in Local Sequence Alignment Algorithm and Its Parallel Acceleration Research, Military Medicine, 36(7), 556-560.
13. Sun M., X. Zhou, F. Yang, K. Lu (2014). Bwasw-cloud: Efficient Sequence Alignment Algorithm for Two Big Data with MapReduce, Applications of Digital Information & Web Technologies, 213-218.
14. Wang C., X. Li, P. Chen, A. Wang, X. Zhou, H. Yu (2015). Heterogeneous Cloud Framework for Big Data Genome Sequencing, IEEE/ACM Transactions on Computational Biology & Bioinformatics, 12(1), 166-178.
15. White T., D. Cutting (2009). Hadoop: The Definitive Guide, O'Reilly Media Inc. Graven Stein Highway North, 215(11), 1-4.
16. Xu B., C. Li, H. Zhuang, J. Wang, Q. Wang, X. Zhou (2017). Efficient Distributed Smith-Waterman Algorithm Based on Apache Spark, IEEE International Conference on Cloud Computing, 608-615.
17. Xue Q. F. (2015). Researches on Parallel Algorithm S for DNA Sequence Alignment, Shanghai: Shanghai University. (in Chinese)
18. Zhao M., W. P. Lee, E. P. Garrison, G. T. Marth (2013). SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications, PloS ONE, 8(12), 2138-2143.
19. Zhu Y. Q., D. J. Niu, T. Cai, H. E. Yao (2016). Test and Analysis of Big Data System in Different Network Environment, Journal of Jiangsu University (Natural Science Edition), 37(4), 429-437. (in Chinese)
20. Wang C. (2015). A Modified Machine Learning Method Used in Protein Prediction in Bioinformatics, Int J Bioautomation, 19(1), 25-36.

Yanfeng Liu, M.Sc. Student
E-mail: 2424568699@qq.com

Yanfeng Liu is a Master degree candidate in Software Engineering at the College of Data Science and Application, Inner Mongolia University of Technology, Hohhot, Inner Mongolia, China. Liu has a Bachelor's degree in Software Engineering from Qilu University of Technology, China. Her research interests include software engineering, cloud computing and big data processing, etc.

Leixiao Li, Ph.D. Student
E-mail: llxhappy@126.com

Leixiao Li is a Ph.D. candidate in Agricultural Information Technology at the College of Computer and Information Engineering, Inner Mongolia Agricultural University, Hohhot, Inner Mongolia, China. Li has a Master's degree in Computer Application Technology from Inner Mongolia University of Technology, China. His research interests include data mining, cloud computing and big data processing, etc.

Prof. Jing Gao, Ph.D.
E-mail: gaojing@imau.edu.cn

Jing Gao is a Professor at the College of Computer and Information Engineering, Inner Mongolia Agricultural University, Hohhot, Inner Mongolia, China. Gao has a Ph.D. in Computer Software and Theory from Beihang University, China. Her research interests include computer software and theory, distributed computing, bioinformatics, big data processing, etc.