ScSpark: Low-Redundancy Disk Access and High-Performance Tool for the Single-Cell RNA Sequenceing Data Processing

1st Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

4th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

5th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

6th Given Name Surname
dept. name of organization (of Aff.)
name of organization (of Aff.)
City, Country
email address or ORCID

***Abstract*—High-throughput single-cell RNA sequencing (scRNA-seq) data processing pipelines typically integrate multiple modules to transform raw scRNA-seq data to gene expression matrices, including barcode processing, sequence quality control, genome alignment and transcript counting. With the drastic increase in scRNA-seq data size, file input/output (IO) has become a bottleneck of the processing speed of available tools. In this study, we took advantage of Apache Spark's in-memory computing and inherent scalability to develop a new Java-based scRNA-seq data processing pipeline, named scSpark. To reduce unneccessary disk access while reading FASTQ files and writing SAM files, we used Java Native Interface (JNI) to deliver FASTQ Resilient Distributed Datasets (RDD), and then retrieved genome mapping results to SAM RDD. By allocating scRNA-seq data and processing tasks to computer cluster, the scSpark toolkit can significantly reduce disk access for saving and loading temporary results. To test the performance of scSpark, we built a spark cluster on Aliyun, and evaluated its computational performance and biological analysis robustness with several state-of-the-art data processing pipelines. The results indicate that scSpark is more efficient and more scalable than the other available tools.**

***Index Terms*—scRNA-seq data processing, Apache Spark, cloud computing**

I. Introduction

Single cell is the fundamental unit of a living organism. In the era of precision medicine, exploring the gene expression at single cell level has become crucial. In the past decade, RNA-seq has been widely used to study gene expression patterns in large-scale biological samples. However, the resolution of bulk RNA-seq could only reach the average level of cell populations. Single-cell RNA sequencing (scRNA-seq) essentially reveals the transcriptional status at single cell level, which provides the basis for subsequent bioinformatics analysis [1].

High-throughput (HT) scRNA-seq protocols, which enable transcript sequencing of thousands of cells simultaneously in a single experiment, have emerged as powerful tools to identify and characterize cell types in complex and heterogeneous tissues [2]. To allow the reads to be demultiplexed after the cells being assembled for sequencing [3], two important oligonucleotide barcodes, namely, cell barcode (CB) and unique molecular identifier (UMI) are introduced in scRNA-seq [4] [5]. CB, which assigns different sequences to each cell for transcript source retrieval after sequencing, greatly improves the throughput and reduces the cost of scRNA-seq [6] [7]. In addition, UMIs are random oligonucleotide barcodes that are used in scRNA-seq experiments [8] [9] to distinguish redundant transcript generated from PCR amplification [10].

In scRNA-seq experiment, it is the usage of these multi-level barcodes that presented additional challenges for data processing, which was quite different from traditional bulk RNA-seq and low-throughput scRNA-seq experiments. In recent years, researchers have developed multiple scRNA-seq data processing tools, typically implementing steps including barcode processing, sequence quality control, genome alignment and transcript counting to convert raw scRNA-seq data into a gene expression matrix for further downstream analysis. The first step is to extract barcodes from a predesigned nucleotide site for different scRNA-seq protocols. Then, FASTQ reads with low quality nucleotides are filtered based on user-defined thresholds. Subsequently, the remaining FASTQ reads are mapped to the reference genome using the splice-aware aligner, such as STAR [11] or HISAT2 [12]. Finally, the reads are assigned to genes and count matrices for UMIs are generated. [13].

Nowadays, there are many tools to preprocess scRNA-seq data, among which the most popular ones inlude CellRanger [14], UMI-tools [10], STARsolo [15], etc. CellRanger

is a highly integrated data processing software tool tailored by 10X Genomics for scRNA-seq data analysis, and it presented best performance on the machine with enough CPU and memory resources for big datasets, while performed slightly poor for small datasets [16]. In addition, UMI-tools (https://github.com/CGATOxford/UMI-tools) is an open source tool with an impressively clear process. And it had relatively higher trascript quantification accuracy compared with other tools in a previous evalution [16]. Furthermore, STARsolo [15] has improved the parallelism of sequence mapping and counting procedures, and achieved good performance on a single machine. Great improvement has been achieved recently for the development of scRNA-seq data processing tools. However, all the avaliable scRNA-seq data processing tools can only run on a single machine with limited running speed, and cannot be extended to multiple nodes.

In recent years, there have been many studies focusing on the optimization of next generation sequencing (NGS) data processing using big data framework like Hadoop (https://hadoop.apache.org) and Spark (https://hadoop.apache.org). Due to higher efficiency in utilizing memory computing, Spark is a better big data computing framework than Hadoop's MapReduce [17], [18]. SparkBWA [19] and GPF [20] are good frameworks with Spart for improving the efficiency of NGS data processing. Falco [21] has tried to use Spark in scRNA-seq data preprocessing, but it has two limitations.

Firstly, Falco can't deal with CBs and UMIs, so it is incompatible with any HT scRNA-seq protocol. Secondly, Falco only uses Spark to simply concatenate the aligner and the feature quantitative softwares, which does not reduce the amount of disk reads and writes during alignment and transcript counting. Falco's operations have lots of redundancy disk access which causes its insufficient utilization of Spark's in-memory computation well.

The increase of scRNA-seq datasets require more efficient and faster data processing tools. However, the current scRNA-seq data preprocessing tools were designed without considering of the scalability, which could only run on a single computer and could not be extended to a cluster. From UMI-Tools and CellRanger to STARsolo, parallelism and performance has increased; However, due to the limit of a single machine, all of them have no scalability . The traditional single machine serial processing mode has been unable to meet the demand of computing and storage resources for super-large scale scRNA-seq data. Different with Falco, we enhanced STAR's file I/O interface to reduce redundant disk access. In the end, to achieve parallel and in-memory computing, scSpark implemented transcript counting by combining Spark function and our method that can distributed cumpute the result and generated by a graph algorithm. Our work utilized Spark system architecture to develop a quick and scalable scRNA-seq data preprocessing tool.

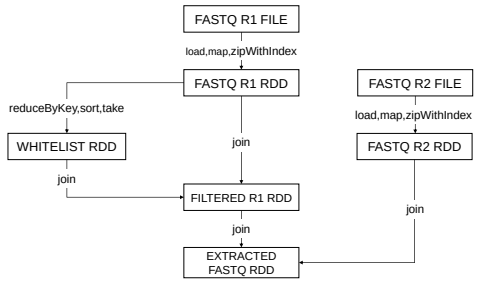

Fig. 1. An overview of sequence quality control module based on Spark.

II. Method

A. Overview

ScSpark implemented barcode and UMI correct, sequence quality control, genome alignment, and transcript counting based on Spark framework and UMI-tools. ScSpark run program across the cluster and default cache data in memory. We implemented barcode and UMI correct when cluster load data and abstract them to RDD. And we developed a high concurrency Spark version sequence quality control. After that, scSpark process STAR's program on each node in the cluster. In the end, scSpark grouped each cell's record and distributed conut the result.

B. Simple barcode and UMI correct

We override FASTQ hadoop loading api (https://github.com /HadoopGenomics). Large FASTQ files can be loaded in each node's memory parallel and the loading speed doesn't limit by single node disk access speed. In this step, scSpark split barcode and UMI when scSpark load FASTQ file to FASTQ RDD. ScSpark splits FASTQ R1's sequence to two part, use barcode as RDD's key and use others information as RDD's value.

C. Spark version sequence quality control

As shown in Fig 1, the sequence quality control step typically consists of two main components, i)generating a whitelist including identities of the true cell barcodes is unknown and ii)filtering FASTQ R1 with the whitelist and extracting FASTQ R2 according to filtered FASTQ R1's index. After FASTQ R1 RDD generated, we calculated the number of occurences for each CB and selected the most frequent CBs, named whitelist RDD, as intermediate results. By using reduceByKey function, each barcode could be counted parallelly in each split of FASTQ R1 RDD and we could merge each split's result in the end. After that we can use the result to generate whitelist RDD by Spark's sort and take function, both of which can parallelly compute in the cluster. The filtered FASTQ R1 RDD could be easily got by using the join function to find which read's cell barcode is in the whitelist RDD. And then we used filter function to extract FASTQ R2 RDD. The last step in sequence quality control was using join function again to

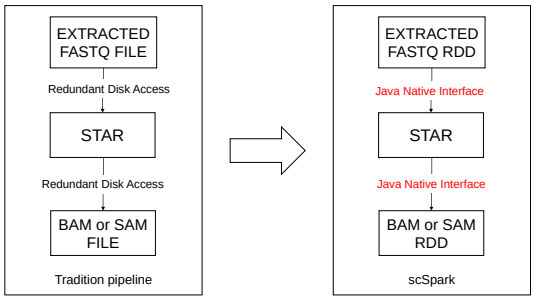

Fig. 2. New way to implement STAR's interface.

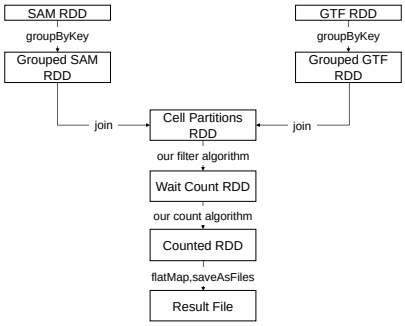

Fig. 3. An overview of Spark version count.

combine filtered FASTQ R1 RDD and FASTQ R2 RDD with the same index.

D. Eleminate redundancy disk access in aligner step

In previous pipeline, STAR needs to load extracted FASTQ R2 file and FASTQ R1 file. This way will generate unnecessary disk access. We choose STAR as our aligner and modify STAR's load FASTQ way to avoid loading extracted FASTQ R2 and FASTQ R1 twice. As shown in Fig 2, we utilized Java Native Interface to transfer extracted FASTQ R2 RDD's data to STAR program, and then run the STAR process. The STAR aligner was encapsulated as a shared object and invoked the shared object as our aligner. To overcome imbalance problem of each node's data volume, we repartitioned the extracted FASTQ R2 RDD before transfer FASTQ RDD to STAR. Each node could run STAR parallelly with counterpoise data volume. We also found if node's in memory was enough, we could start up more than one STAR's in one node to achieve better parallel computing. After align, STAR's output is SAM file or BAM file that will generate more disk access waste than STAR's input. Here, We used Java Native Interface again to transfer STAR results and directly abstract them as SAM RDD.

E. Count with multi-node

As shown in Fig 3, to take advantage of multi-node's compute capability, we used a new way to implement the count step. Except for SAM RDD that were generated in the previous step, we loaded GTF file to memory and abstract them as GTF RDD. After that scSpark grouped SAM RDD and GTF RDD according to their cell name. Then we used flatMap function to parallelly compute each cell's read count. In the end, we collected results in all nodes, and the result files would produce the only output disk access in the whole program. ScSpark breaks the limitation of one node's computing and achieved large scalability.

F. Experiment design

We used Apache Spark version 2.1.0 as our in memory computing environment. And three example scRNA-seq datasets generated by 10X Genomics on their platform was used in our experiments. Three datasets all come from 10xgenomics datasets (support.10xgenomics.com). Dataset1, 10X Genomics v3 10k peripheral blood mononuclear cell (PBMC) from a healthy donor, which contain 640 millions records. Dataset2, 10X Genomics 10k Rat PBMCs Multiplexed from a Wistar rat strain (www.10xgenomics.com/resources/datasets/10-k-rat-pbm-cs-multiplexed-2-cm-os-3-1-standard-6-0-0), which contain 289.3 millions records. Dataset3, 10X Genomics 10k Monkey PBMCs Multiplexed from a Rhesus monkey strain (www.10xgenomics.com/resources/datasets/10-k-monkey-pbm-cs-multiplexed-2-cm-os-3-1-standard-6-0-0), which contain 261.5 millions records. To compare with tradition pipelines, we tested three datasets in UMI-tools, CellRanger, STARsolo and scSpark's performance. And then, we tested each step's spend time, due to CellRanger isn't an open source software. And STARsolo implemented pipeline in a different way, we choose UMI-tools as scSpark each step's performance baseline. After that, in align step, four pipelines all based on STAR's program. So we tested STAR and scSpark's mapping speed to prove scSpark not only get greatly improve in same CPU cores compare with STAR, but also can get near linear improve when CPU cores increase. Except performance evaluate, we use 16 CPU cores as tradition pipeline's scalability baseline and use 64 CPU cores as scSpark's scalability baseline to prove scSpark's scalability better than tradition pipelines.

III. Results

We evaluated the speed, scalabilty and compare each step's scalability of scSpark in this section.

A. Efficiency evaluation

Limited by single machine's CPU cores, we only take 64 cores in traditional pipeline test as the traditional pipeline's performance. Table I gives a summary of our program's performance and compare with tradition pipeline's performance. We can see scSpark's speed is much more quick than any tradition pipeline in same CPU cores environment. And scSpark can get improve when the cluster's CPU cores number increase. We recorded each step's process time to know the reason of the speed's improvement. As table II shown, scSpark is much faster than the UMI-tools in any single step. Contrast with

TABLE I
PREVIOUS PIPELINE AND SCSPARK'S PERFORMANCE COMPARISIONE

System	Dataset	Cores	Spend time(s) 100M reads	640M reads
UMI-tools	1	64 cores	7254	44160
CellRanger	1	64 cores	6000	11700
STAR-solo	1	64 cores	5820	8100
scSpark	1	4*16 cores	355	-
	1	16*16 cores	-	841
UMI-tools	2	64 cores	7254	44160
CellRanger	2	64 cores	6000	11700
STAR-solo	2	64 cores	5820	8100
scSpark	2	4*16 cores	326	-
	2	16*16 cores	-	841
UMI-tools	3	64 cores	7254	44160
CellRanger	3	64 cores	6000	11700
STAR-solo	3	64 cores	5820	8100
scSpark	3	4*16 cores	391	-
	3	16*16 cores	-	841

TABLE II
PERFORMANCE COMPARISION FOR EACH STEP

System	Dataset	Cores	Barcode and UMI correct, filter	align	count
UMI-tools(100 million reads)	1	64 cores	9720	600	1740
UMI-tools(640 million reads)	1	64 cores	33600	2160	8400
scSpark(100 million reads)	1	4*16 cores	130	132	93
scSpark(640 million reads)	1	16*16 cores	81	447	313
UMI-tools(100 million reads)	2	64 cores	9720	600	1740
UMI-tools(289.3 million reads)	2	64 cores	33600	2160	8400
scSpark(100 million reads)	2	4*16 cores	137	117	72
scSpark(289.3 million reads)	2	16*16 cores	81	447	313
UMI-tools(100 million reads)	3	64 cores	9720	600	1740
UMI-tools(261.5 million reads)	3	64 cores	33600	2160	8400
scSpark(100 million reads)	3	4*16 cores	129	227	35
scSpark(261.5 million reads)	3	16*16 cores	81	447	313

UMI-tools, We could see scSpark gets great improve in the first two steps, barcode correct and filter. This step's improve comes from using in memory trait to reduce disk access times and take advantage of Spark's high concurrency feature.

B. Scalability Evalution

ScSpark's second advantage is that it can get near linear improvement when the cluster's CPU core number improve. To test the scalability of the tradition pipeline and scSpark, we use 16 CPU cores performance as tradition pipeline's baseline and 64 CPU cores performance as scSpark's baseline. We use a small sample which have 100 millions records to compare speedup between our program and tradition pipeline. As shown in Fig 5, We found when CPU cores number increase our program can get near linear speedup and tradition pipeline's

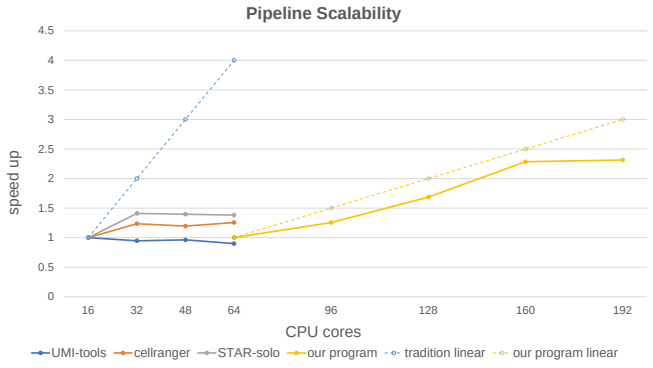

Fig. 4. An overview of Spark version filter.

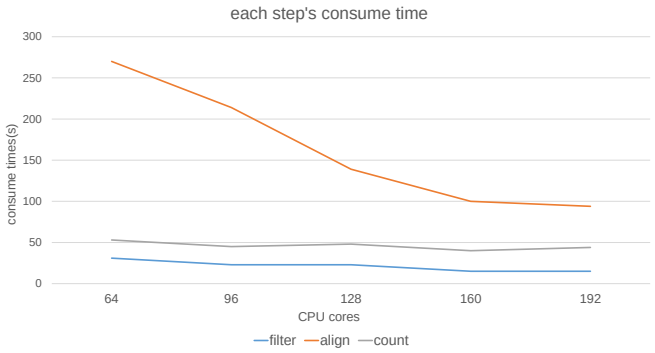

Fig. 5. Each step consumes time.

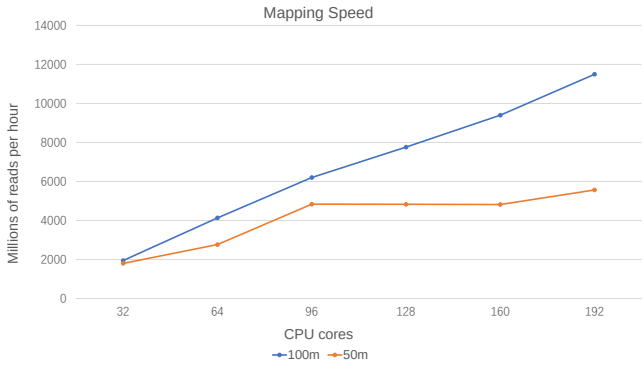

Fig. 6. Invoked STAR's mapping step.

speedup far below linear speedup. And we also found if the CPU cores number exceeds a ceiling, both tradition pipeline and scSpark will speedup nearly stop. UMI-tools isn't showing any scalabiliy, even a little slow down when the CPU cores number increase. STARsolo and CellRanger shows a little improve when the CPU cores increases to 32 from 16, but quickly stop speedup. But scSpark ceiling is much higher than tradition pipeline. All of it explain that our program more scalability than all tradition pipeline.

C. Comparsion each step performance's increase

We also record each step's process time to know the reason of our improve and where can improve in future. As shown in Fig 5, we found our program's scalability most come from the align step and the align consumes most time in the whole program. To find the reason why the align step is more scalability than the other step and why the scalability have a ceiling, we count how much records scSpark's STAR program can process. We can find in Fig 6, STAR's mapping speed is influenced by the data volume size and small volume dataset will lost its scalability earier than large volume dataset.

Then we test STAR program, and find the mapping speed is influenced by data volume. As Table III shown, STAR mapping speed will increase when the data volume increase. So

TABLE III
PROCESSING SPEED W.R.T. DATA VOLUME

Volume (million reads)	10	50	100	640
Speed (million reads per hour)	250.28	503.83	503.02	950

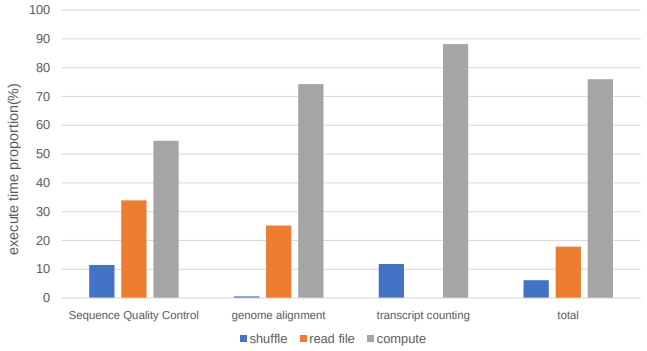

Fig. 7. Each step consumes time.

when we get scalability by increase partition, each partition's read number decrease will limit the scalability increase.

D. Performance Analysis

We tested performance to find bottleneck of our scSpark. We used 640 millions records of FASTQ data to evaluate two aspect of our scSpark. First we compared scSpark spend in network shuffle, disk access and compute time proportion to find whether network shuffle or disk access occupies too much time. Second we tested CPU and memory usage, to ensure which resources will be scSpark's bottleneck.

1) Network and disk behavior: We computed network time by summing up the time that our scSpark shuffle data in multi machine. Disk access comes from loading FASTQ, STAR's index and GTF files. The ideal situation is tasks did not waste any time in disk access and network shuffle. We found that scSpark's computing time occupies most execute time. Except STAR's index file, all file's disk access distribute to each node that improve whole system's loading speed. And we found the time that waste in shuffling doesn't occypy too much time.

2) CPU and memory usage: We monitored scSpark's CPU and memory usage during processing. As Fig 8 shown, in sequence quality control step, scSpark highly exploited each node's multi cores CPU to achieve speedup. Convention pipelines single thread solution's CPU usage is much lower than scSpark. And we found that scSpark's boundary much comes from genome alignment. Because we invoked STAR as our alignment tool, and STAR's program naturally occupy most proportion of memory in this step.

3) Biological verification: Our scSpark is developed based on UMI-tools. And UMI-tools accuracy was fully verified. This section we uses the gene expression matrix obtained by scSpark and UMI-tools under the same dataset to perform downstrema analysis of scRNA-seq data. And then we compared two tools transcript analysis's result and cell cluster

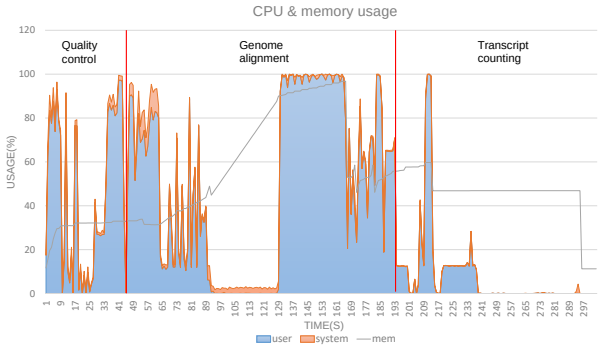

Fig. 8. CPU and Memory usage of scSpark.

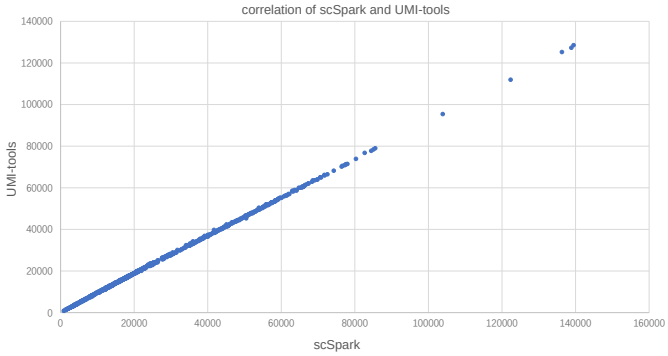

Fig. 9. correlation of scSpark and UMI-tools.

analysis's result to verify the correlation between scSpark and UMI-tools. Under hgmm-1k-v3 dataset, we used scSpark and UMI-tools to get gene express matrix, compared two tools' result, and computed their correlation. As Fig 9 shown, for each after processing cell barcode, their gene expression matrix approximate fit $y = x$, R^2 closed to 0.9998. Furthermore, we used Seurat to print tSNE picture which based on gene expression matrix generated by ScSpark and UMI-tools. And as Fig 10 shown, tSNE cell clustering result showed high corrleation between ScSpark and UMI-tools.

IV. CONCLUSION

In this paper, we proposed a way which utilize Apache Spark's in memory compute trait and get considerable speedup and scalability. Our scSpark can take advantage of multi machine compute capacity to speedup all step and eliminate redundant disk access.

Except performance improve, our scSpark also show much more scalability than any tradition pipeline which closes to linear improve. Our scSpark not only can improve each partition process STAR program's mapping speed, but also can improve scalability by increasing partition number if the resource is sufficient.

And we also found our scSpark's scalability improve has a ceiling. The reason is that our scSpark invokes STAR's

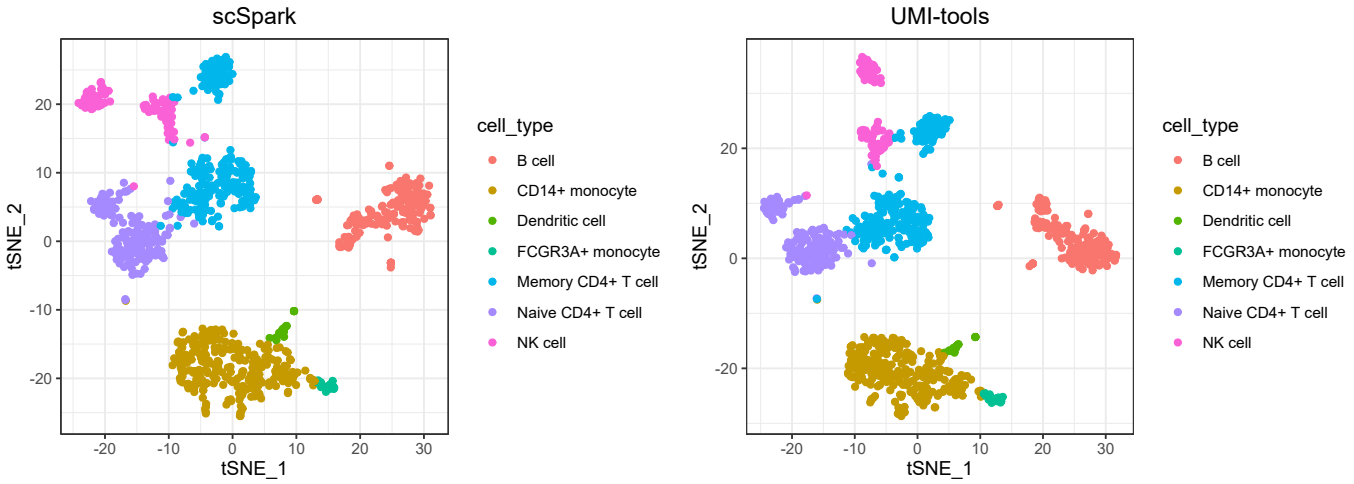

Fig. 10. tSNE picture based on scSpark and UMI-tools' gene expression matrix.

mapping speed influenced by loading index time. And if data volume is tiny, the influence occupies a large proportion of whole the STAR program process time.

REFERENCES

[1] E. Papalexi and R. Satija, "Single-cell RNA sequencing to explore immune cell heterogeneity," *Nature Reviews Immunology*, vol. 18, pp. 35–45, 2018.

[2] X. Zhang, T. Li, F. Liu, Y. Chen, J. Yao, Z. Li, Y. Huang, and J. Wang, "Comparative analysis of droplet-based ultra-high-throughput single-cell RNA-seq systems," *Molecular cell*, vol. 73, no. 1, pp. 130–142.e5, 2019.

[3] L. Tian, S. Su, X. Dong, D. Amann-Zalcenstein, C. Biben, A. Seidi, D. J. Hilton, S. H. Naik, and M. E. Ritchie, "scPipe: A flexible R/Bioconductor preprocessing pipeline for single-cell RNA-sequencing data," *PLoS Computational Biology*, vol. 14, no. 8, 2018.

[4] A. Rosenberg, C. M. Roco, R. A. Muscat, A. Kuchina, P. Sample, Z. Yao, L. T. Graybuck, D. J. Peeler, S. Mukherjee, W. Chen, S. Pun, D. L. Sellers, B. Tasic, and G. Seelig, "Single-cell profiling of the developing mouse brain and spinal cord with split-pool barcoding," *Science*, vol. 360, pp. 176 – 182, 2018.

[5] J. Cao, J. S. Packer, V. Ramani, D. A. Cusanovich, C. Huynh, R. Daza, X. Qiu, C. Lee, S. Furlan, F. Steemers, A. C. Adey, R. Waterston, C. Trapnell, and J. Shendure, "Comprehensive single cell transcriptional profiling of a multicellular organism by combinatorial indexing," *bioRxiv*, 2017.

[6] E. Z. Macosko, A. Basu, R. Satija, J. Nemesh, K. Shekhar, M. Goldman, I. Tirosh, A. Bialas, N. Kamitaki, E. M. Martersteck, J. J. Trombetta, D. Weitz, J. Sanes, A. Shalek, A. Regev, and S. McCarroll, "Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets," *Cell*, vol. 161, pp. 1202–1214, 2015.

[7] A. M. Klein, L. Mazutis, I. Akartuna, N. Tallapragada, A. Veres, V. Li, L. Peshkin, D. Weitz, and M. Kirschner, "Droplet barcoding for single-cell transcriptomics applied to embryonic stem cells," *Cell*, vol. 161, pp. 1187–1201, 2015.

[8] T. Kivioja, A. Vähärautio, K. Karlsson, M. Bonke, M. Enge, S. Linnarsson, and J. Taipale, "Counting absolute numbers of molecules using unique molecular identifiers," *Nature Methods*, vol. 9, pp. 72–74, 2012.

[9] P. G. Camara, "Methods and challenges in the analysis of single-cell RNA-sequencing data," *Current Opinion in Systems Biology*, vol. 7, pp. 47–53, 2018.

[10] T. Smith, A. Heger, and I. M. Sudbery, "UMI-tools: modeling sequencing errors in unique molecular identifiers to improve quantification accuracy." *Genome research*, vol. 27, no. 3, pp. 491–499, 2017.

[11] A. Dobin, C. A. Davis, F. Schlesinger, J. Drenkow, C. Zaleski, S. Jha, P. Batut, M. Chaisson, and T. R. Gingeras, "STAR: ultrafast universal RNA-seq aligner," *Bioinformatics*, vol. 29, no. 1, pp. 15–21, 2013.

[12] D. Kim, B. Langmead, and S. Salzberg, "HISAT: a fast spliced aligner with low memory requirements," *Nature Methods*, vol. 12, no. 4, pp. 357–360, 2015.

[13] S. Parekh, C. Ziegenhain, B. Vieth, W. Enard, and I. Hellmann, "zUMIs - A fast and flexible pipeline to process RNA sequencing data with UMIs," *GigaScience*, vol. 7, 2018.

[14] G. X. Y. Zheng, J. M. Terry, P. Belgrader, P. Ryvkin, Z. W. Bent, R. Wilson, S. B. Ziraldo, T. D. Wheeler, G. McDermott, J. Zhu, M. Gregory, J. Shuga, L. Montesclaros, J. Underwood, D. A. Masquelier, S. Y. Nishimura, M. Schnall-Levin, P. W. Wyatt, C. M. Hindson, R. Bharadwaj, A. Wong, K. Ness, L. Beppu, H. Deeg, C. McFarland, K. Loeb, W. Valente, N. G. Ericson, E. Stevens, J. Radich, T. Mikkelsen, B. Hindson, and J. Bielas, "Massively parallel digital transcriptional profiling of single cells," *Nature Communications*, vol. 8, 2017.

[15] A. Blibaum, J. Werner, and A. Dobin, "STARsolo: single-cell RNA-seq analyses beyond gene expression," *F1000Research*, vol. 8, 2019.

[16] M. Gao, M. Ling, X. Tang, S. Wang, X. Xiao, Y. Qiao, W. Yang, and R. Yu, "Comparison of high-throughput single-cell RNA sequencing data processing pipelines," *Briefings in Bioinformatics*, vol. 22, no. 3, 2021.

[17] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012, pp. 15–28.

[19] J. Abuín, J. C. Pichel, T. Pena, and J. Amigo, "Sparkbwa: Speeding up the alignment of high-throughput dna sequencing data," *PLoS ONE*, vol. 11, no. 5, 2016.

[20] X. Li, G. Tan, B. Wang, and N. Sun, "High-performance genomic analysis framework with in-memory computing," *ACM SIGPLAN Notices*, vol. 53, no. 1, pp. 317–328, 2018.

[21] A. Yang, M. Troup, P. Lin, and J. Ho, "Falco: a quick and flexible single-cell RNA-seq processing framework on the cloud," *Bioinformatics*, vol. 33, no. 5, pp. 767–769, 2017.