International Journal of Hybrid Information Technology
Vol.7, No.4 (2014), pp.91-100
http://dx.doi.org/10.14257/ijhit.2014.7.4.09

Parallel Compression and Decompression of DNA Sequence Reads in FASTQ Format

Jingjing Zheng[1,*] and Ting Wang[1, 2]

[1,]Parallel Software and Computational Science Lab, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China*
[2]Research Center of Parallel Software Cloud Application, Institute of Software Application Technology, Guangzhou & Chinese Academy of Sciences, Guangzhou 511458, China
[*]sdzjjing@163.com

Abstract

Large volumes of short reads of genomic data are generated by high-throughput sequencing instruments. The FASTQ format is widely accepted as the input format of genomic reads and has presented challenges in data storage, management, and transfer. The performance of this type of serial algorithms such as G-SQZ and DSRC is limited by the single processor and the memory in a single computer. Utilizing data parallelism, the circular dual queues of buffers, memory mapping integrated with superblocks, pipeline parallelism with multi threads, and so on, we present the parallel compression and decompression methods for DNA sequence reads in FASTQ format based on the parallel computer architectures of the cluster and the SMP. Experimental results for the parallel DSRC algorithm clearly show the efficiency of using the powerful computing resources from multi computing nodes and multi cores of each node. The speedups vary from 46 to 62 for parallel compression and vary from 40 to 58 for parallel decompression by using 10 nodes of the cluster in Tianhe-1A super computer. Test results on the SMP machine are also pleasant. The methods could be applied to any serial compressing algorithms of DNA sequence reads in FASTQ format only if they have the traits of index and superblocks.

Keywords: *High-performance Computing, Data Intensive Computing, Parallel Compression*

1. Introduction

Large volumes of short reads of genomic data are generated by high-throughput sequence instruments. The FASTQ format is the widely accepted input format of genomic reads because compared with other formats (such as the FASTA format), it could contain the annotations (like quality scores denoting uncertainties in sequence identification processes), read identifications and other descriptions (*e.g.*, unique instrument name) at the same time. However, this characteristic of FASTQ format has presented challenges in data storage, management, and transfer.

Considering the compression of the FASTQ file, the G-SQZ [1] and DSRC [2] are two important compression and decompression algorithms proposed in recent years with excellent performance. These two algorithms both use the index to locate and access the specific reads quickly without decoding the whole compressed file. What’s more, the DSRC algorithm divides the whole file data into many superblocks, and the G-SQZ algorithm could also use the superblocks without any difficulty.

Nowadays the size of genomic reads which need to be analyzed has already reached to Terabytes. The above G-SQZ and DSRC algorithms are both serial and the performance could be limited by the single processor and the memory in a single computer, which would affect the processing and analyzing of these mass data. With the rapid development of high performance computing, utilizing the powerful computing performance coming from the multi computing nodes and the multi cores of each node could real-time compress and decompress these data, improve the processing speed and could promote more applications of these genomic data. Unfortunately, up to now, the parallel algorithms about the above serial algorithms have not been proposed.

We propose the methods for parallel compression and decompression of DNA sequence reads in FASTQ format, which could change the above serial G-SQZ and DSRC algorithms into parallel algorithms. Most importantly, these methods could also be applied to other serial compressing algorithms of DNA sequence reads in FASTQ format only if they have the traits of index and superblocks. The experimental results of the parallel DSRC algorithm which combines these proposed methods with DSRC algorithm show that it could achieve performance speedups of up to 40~62x in 10 nodes on cluster of Tianhe-1A with the parallel programming tool of MPI combined with OpenMP. Test results on the SMP machine are also pleasant.

2. Methods

Aiming at the DNA sequence reads in FASTQ format, we propose parallel compression and decompression methods based on the parallel computer architectures of the cluster and the SMP. Utilizing data parallelism, the circular dual queues of buffers, memory mapping integrated with superblocks, pipeline parallelism with multi threads, and a two-dimension array recording reading and writing order, the parallel compression and decompression of DNA sequence reads in FASTQ format with multi processes and multi threads within each process become true. These methods could make the best use of the powerful computing resources from the multi computing nodes and the multi cores of each node, and could eliminate the performance limits of single processor and memory when using the serial algorithm. Not only the parallel programming tool of MPI combined with OpenMP could be used, but the tool of MPI combined with Pthreads could be used as well.

In the following subsections we briefly present the techniques used for the parallel compression and decompression methods.

2.1. Data Parallelism

Because FASTQ data can naturally be perceived as ordered collections of records, each consisting of three streams: title information, DNA sequence (read), and quality scores, it is beneficial to allocate the FASTQ data to multi processes. The FASTQ format was fully discussed in paper [3]. Any FASTQ data parser must not read a line starting with '@' as indicating the start of the next record because the '@' marker character may occur anywhere in the quality string, including at the start of any of the quality lines. Most tools output FASTQ files without line wrapping of the sequence and quality string. This means that in most cases each record consists of exactly four lines, which is ideal for a very simple parser to deal with. According to this convention on output of four lines within each record, in the parallel compression, the FASTQ data are divided nearly averagely into multi data sections which could be compressed concurrently in multi processes. Based on the number of the compressing processes and the total FASTQ file size, the allocated data size of each process could be computed. In addition, the data start could be computed in each process. However, this start position might not be the start of a record. The first '@'-started line after the above computed position should be found further. The second line closely after the above '@'-started line should also be found. If this second line doesn't start with '@', the first '@'-started line would be the tile line, and otherwise, this second line would be the tile line. The start of this tile line would be the exact start in this FASTQ file in this process. In this way, the processing range of the assigned data in each process is computed precisely, which lays the foundation of the data parallelism in compression. This data assigning and paralleling methods get rid of the communications between different processes and different nodes; therefore, the parallel efficiency and speedup could be improved. Each process forms a separate compressed file.

In the parallel decompression, the number of processes is determined by the number of the compressed file, and the sequence of the decompressed file is the same as the compressed file. Each decompressing process has a separate decompressed file. There is also no communication between different processes and different nodes. The decompressing process could obtain the exact compressed data location of each superblock via the index in the compressed file, which could facilitate the data parallelism in each decompressing process.

2.2. Circular Dual Queues of Buffers

Whether in compressing process or in decompressing process, each process contains multi threads: one reading thread, one writing thread, and several working threads (the number of working threads could be set). The circular dual queues of buffers are the linkers between these threads. For the sake of easy description, we call the circular dual queues of buffers a circular-queues-linker. Each working thread has two circular-queues-linkers: one input circular-queues-linker, one output circular-queues-linker. These two types of circular-queues-linker have slightly different structures according to the difference of the data stored in them for the purpose of use. Despite of this, they have the same working mechanism. This mechanism has been showed in the Figure 1.

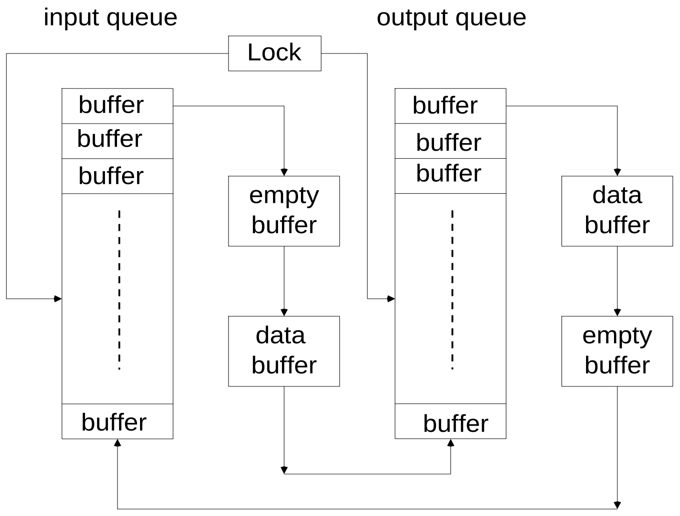

Figure 1. Circular Dual Queues of Buffers

Each circular-queues-linker has two queues: one input queue, and one output queue. Within each circular-queues-linker, the input queue and the output queue have the same structure. The work mechanism of the circular-queues-linker is as follows:

(1) Initialization: instantiate the input queue as a queue with certain number of empty buffers (Each buffer has the same structure) and the output queue as zero buffer.

Then the following steps progress circularly:

(2) An empty buffer is popped up from the front of the input queue and then is filled with data by the thread which uses this buffer.

(3) The above buffer with filled data is pushed to the back of the out queue.

(4) A buffer with data is popped up from the front of the output queue and then the data in the buffer is processed by the thread which uses them.

(5) After the data in the picked buffer in step (4) have been processed, this buffer is cleaned and pushed to the back of the input queue.

In order to avoid the conflict in using this circular-queues-linker between different threads, we use the lock. In addition, the task of each buffer in each queue is to store superblock data (the original data or the compressed data of a superblock) and other information (such as the data length, the No. of the superblock, *et. al.*)

This mechanism effectively alleviates the delay caused by the mismatch of processing speeds between different types of threads, and could increase the parallel efficiency between these threads.

2.3. Pipeline Parallelism with Multi Threads

Whether in compressing process or in decompressing process, each process implements the pipeline parallelism with multi threads: one reading thread, multi working threads, and one writing thread. The Figure 2 gives the frame of this mechanism.

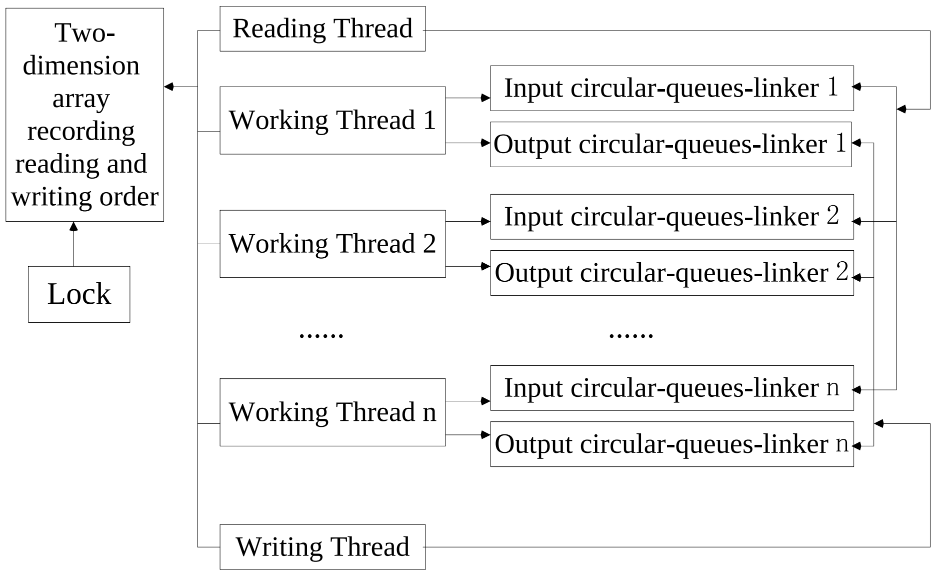
Two-dimension array recording reading and writing order
Lock
Reading Thread
Working Thread 1
Working Thread 2
......
Working Thread n
Writing Thread
Input circular-queues-linker 1
Output circular-queues-linker 1
Input circular-queues-linker 2
Output circular-queues-linker 2
......
Input circular-queues-linker n
Output circular-queues-linker n

Figure 2. Pipeline Parallelism with Multi Threads

Just as we have said, each working thread has two circular-queues-linkers: one input circular-queues-linker, and one output circular-queues-linker. The input circular-queues-linker in each working thread is the data interface between this working thread and the reading thread, and the output circular-queues-linker in each working thread is the data interface between this working thread and the writing thread. In order to guarantee that the order of processed data between these superblocks in the writing thread is the same as the order in the reading thread, a two-dimension array is used to record the reading and writing order by signifying the ID of working thread processing each superblock and whether each superblock has been processed or not. These threads may use the array at the same time, so another lock is used to avoid conflict between these threads.

Using the above components, these three types of threads conduct the pipeline parallelism in each process. The concrete steps are as follows:

(1) The reading thread continually read the data of a superblock (In compression, they would be the original data, and in decompression, they would be the compressed data.), and it search circularly the input queue of each input circular-queue-linker in each working thread until an empty buffer is found in a specific working thread. Then this buffer is popped out from the front of the input queue and is filled with the above data of a superblock. After that, this buffer is pushed to the back of the out queue of the input circular-queue-linker in the above specific working thread. The above procedure carries out continually until all the assigned data of this process are finished reading out. The two-dimension array records the ID of working thread processing each superblock.

(2) Each working thread continually searches the output queue of its input circular-queue-linker until a buffer is found. Then this buffer is popped out from the front of the out queue, and then the data in it are processed (compressed or decompressed). After that, this buffer is cleaned and pushed to the back of the input queue.

Then the following important step is that this working thread continually searches the input queue of its output circular-queue-linker until an empty buffer is found. Then this buffer is popped out from the front of the input queue, and is filled with the above processed data of a superblock. After that, this buffer is pushed to the back of the out queue of this output circular-queue-linker in this working thread.

The above procedure carries out continually until all the assigned data of this process are finished being processed. After a superblock has been processed, the finishing tag is assigned in the two-dimension array.

(3) According to the order and processing progress of superblocks stored in the two-dimension array, the writing thread continually find the specific data of superblock in a specific working thread sequentially and write these data to a file according to the same order as the reading thread. The writing thread searches the output queue of the output circular-queue-linker in the specific working thread until a buffer is found. Then this buffer is popped out from the front of the out queue, and the data in it are written to the file. After that, this buffer is cleaned and pushed to the back of the input queue. The above procedure carries out continually until all the data of supper blocks in this process are written to the file.

2.4. Memory Mapping Integrated with Superblocks

In the reading thread for the parallel compression and the writing thread for the parallel decompression, voluminous data of DNA sequence reads should be dealt with; therefore, the memory mapping integrate with superblocks is used in order to improve the I/O speed of large file.

In the reading thread of parallel compression, according to the page size of the memory, the size of mapping space, and superblocks of original DNA sequence reads in the FASTQ file, the location of each superblock in the memory mapping space are computed, as well as the time when to release and remap the memory mapping. One evident advantage of memory mapping is that the data could be read and written directly with much less data copy when compared with the I/O functions of fread and fwrite.

In the writing thread of parallel decompression, a memory mapping file with specific sufficient size is created according to the number of the superblocks needed to be decompressed in the process. The decompressed data of each superblock are stored into the memory mapping space in turn with the same order as they are read in the reading thread. During this procedure, based on the page size of the memory, the location of the memory mapping space which is needed to be filled, the size of mapping space, the current offset of the memory mapping file, and the threshold of remapping, when to release and remap the memory mapping are determined, as well as the new threshold of remapping. What should be emphasized is that there is null space in the back of the memory mapping file because this file size is set at the start of this writing thread.

3. Implementation and Results

In order to demonstrate the efficiency of these methods, we implemented the parallel compression and decompression by integrating the proposed methods with the DSRC algorithm via the parallel programming tool of MPI combined with OpenMP. This implementation is called PDSRC (Parallel DSRC). The MPI [4] is used as the tool for the interaction between different processes, and the OpenMP [5] is used for the multi threads in each process. The experimental results of PDSRC on cluster of Tianhe-1A and the SMP machine respectively have shown the efficiency of the methods for parallel compression and decompression.

3.1. Experimental Results on Cluster of Tianhe-1A

The test data comprise some files from 1000 Genomes Project (www.1000genomes.org). The test machine was the cluster of Tianhe-1A super computer [6]. All the computing nodes and I/O nodes are interconnected with high-speed network. Each computing node is deployed with dual Intel Xeon X5670 2.93 GHz processors, 6 cores in each processor and 24GB memory. In addition, each computing node has the SMP (Symmetric Multiprocessor, SMP) architecture. The detailed experiments are described as follows. What should be emphasized is that for all the tests, time measurements were performed with the Unix **time** command and each time value is the average of many tests.

Whether in compression or in decompression, the number of working threads could be set through parameter. The compressing time and decompressing time with different numbers of working threads were compared using the test data SRR099458_2.filt.fastq (25.91 GB) and SRR013951_2.filt.fastq (3.19 GB) in 1 and 10 computing nodes with all the 12 cores of each node. The test results showed that each node with 3 processes and each process with 2 working threads has the least compressing time, and each node with 3 processes and each process with 10 working threads has the least decompressing time. The following tests used these parameters.

The parallel compressing time and decompressing time were tested using different data and different nodes from 1 node to 10 nodes. All the 12 cores of each node were fully used. The test data are three FASTQ files: SRR099458_2.filt.fastq (25.91GB), SRR027520_1.filt.fastq (4.81 GB), and SRR013951_2.filt.fastq (3.19GB). For all tested parallel compression and decompression, we present the parallel speedups for nodes from 1 to 10 nodes and for the three files. The Figure 3 and Figure 4 show the results. We could see that typically the speedup increases with the increase of nodes count, which means that the best speedups are got mostly with the 10 nodes. The best speedups vary from 46 to 62 for parallel compression and vary from 40 to 58 for parallel decompression on the cluster of Tianhe-1A.

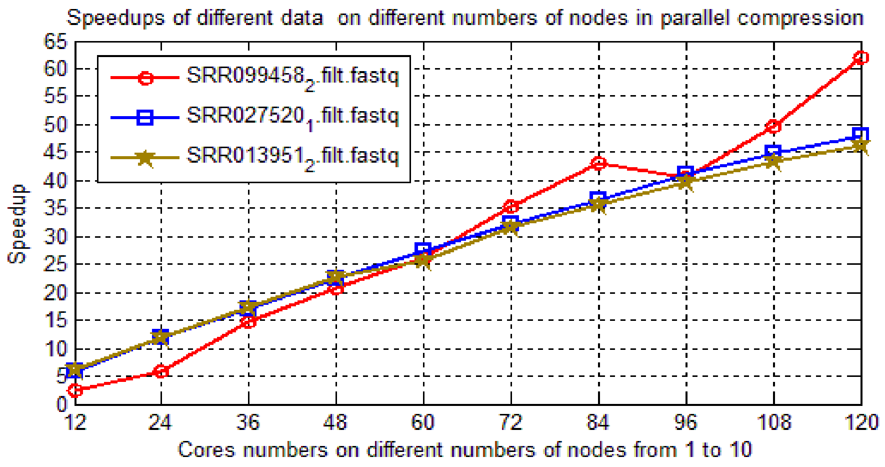

Figure 3. Speedups on Different Numbers of Nodes for Parallel Compression

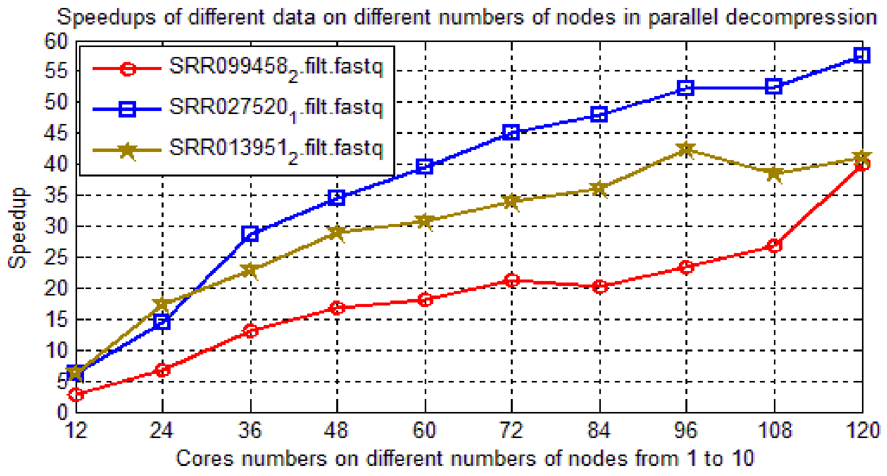

Figure 4. Speedups on Different Numbers of Nodes for Parallel Decompression

3.2. Experimental Results on SMP Machine

The test data and test method are the same as the above test. However, this test machine is SMP architecture, which has 8 Intel Xeon X7550 2.0 GHz processors, 8 cores in each processor and 512 GB memory. What should be emphasized is that this machine uses the Hyper-Threading technology and the above cluster of Tianhe-1A doesn’t use this technology. The detailed experiments are described as follows.

The compressing time and decompressing time with different numbers of working threads were compared by experiments. Experimental results show that each process with 3 working threads has the least compressing time and decompressing time. The following tests used these parameters.

The parallel compressing time and decompressing time were tested using the above three FASTQ files and different numbers of processes from 1 to 25. The different numbers of processes from 1 to 25 use different numbers of cores from 2.5 to 62.5 respectively. For all tested parallel compression and decompression, we present the speedups on different numbers of cores. The figure 5 and figure 6 show the results. We could see that the best speedups vary from 26 to 30 for parallel compression and vary from 13 to 27 for parallel decompression on this SMP machine.

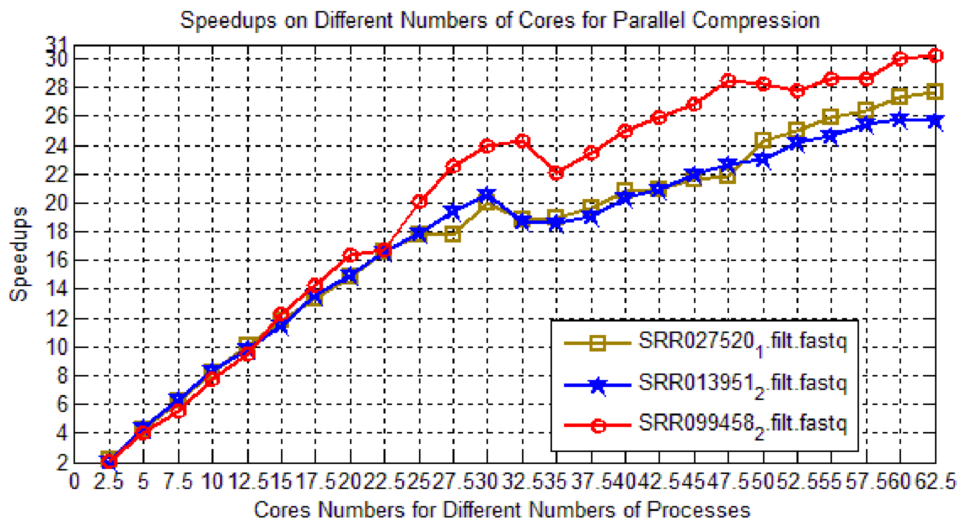

Figure 5. Speedups on Different Numbers of Cores for Parallel Compression

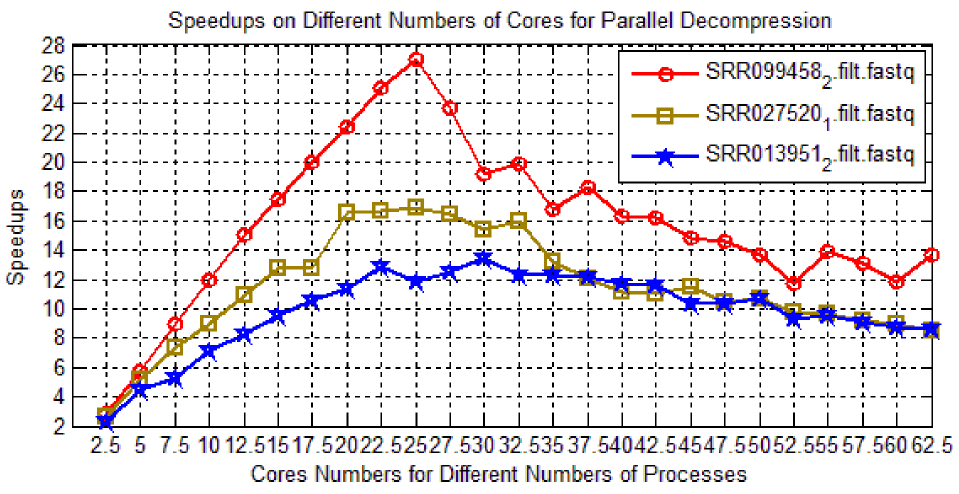

Figure 6. Speedups on Different Numbers of Cores for Parallel Decompression

What should be highlighted here is that the outputs of parallel compression are multi compressed files. Compared with the serial compressed file, the total size of these multi compressed files vary from -0.0014% to 0.0013%, -0.0126% to 0.0088% and -0.0216% to 0.0191% for SRR099458_2.filt.fastq (25.91 GB), SRR027520_1.filt.fastq (4.81 GB) and SRR013951_2.filt.fastq (3.19GB) respectively.

4. Discussion

What should be emphasized here is that these methods could also be applied to other serial compressing algorithms of DNA sequence reads in FASTQ format only if they have the traits of index and superblocks. Because many details of the G-SQZ algorithm can not be inferred from the paper [1] and we were unable to obtain program sources although basically it uses the index mechanism and applies order-0 Huffman coding on combined bases and respective qualities, the parallel G-SQZ was not implemented. However, with the index mechanism of G-SQZ, it is evident that parallel G-SQZ could also be implemented easily using these methods proposed in this paper by integrating the superblocks with the index if we get more details about the G-SQZ. We are sure that some pleasant results would be gotten for the parallel G-SQZ compression and decompression.

5. Conclusion

We presented the parallel compression and decompression methods for DNA sequence reads in FASTQ format based on the parallel computer architectures of the cluster and the SMP. Experimental results for PDSRC clearly showed the efficiency of these parallel methods. The speedups vary from 46 to 62 for parallel compression and vary from 40 to 58 for parallel decompression by using 10 nodes of cluster in Tianhe-1A super computer. Test results on the SMP machine are also pleasant.

These methods could make the best use of the powerful computing resources from the multi computing nodes and the multi cores of each node, and could eliminate the limits of single processor and memory when using the serial algorithm. Not only the parallel programming tool of MPI combined with OpenMP could be used, but also the tool of MPI combined with Pthreads could be used. In addition, the proposed parallel methods could also be applied to other serial compressing algorithms of DNA sequence reads in FASTQ format only if they have the traits of index and superblocks.

Acknowledgements

This paper is supported by the Natural Science Foundation of China (61100073), and the Cloud Computing Technology Development and Industrialization Special of Guangzhou, China (2013Y2-00076).

References

[1] W. Tembe, J. Lowey and E. Suh, “G-SQZ: Compact Encoding of Genomic Sequence and Quality Data”, Bioinformatics, vol. 26, no. 17, **(2010)**, pp. 2192–2194.
[2] S. Deorowicz, and S. Grabowski, “Compression of DNA Sequence Reads in FASTQ Format”, Bioinformatics, vol. 27, no. 6, **(2011)**, pp. 860-862.
[3] 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, **(2010)**, pp. 1767–1771
[4] MPI, http://www.mpich.org/.

[5] OpenMP, http://openmp.org/wp/.
[6] Tianhe-1A Super Computer, https://vpn.nscc-tj.cn/svpn/domain1/www/login/index.htm.

Authors

Jingjing Zheng, obtained her Ph. D. degree of Computer Science and Technology from Institute of Computing Technology, Chinese Academy of Sciences in July 2009. Now she works in Parallel Software and Computational Science Lab of Institute of Software, Chinese Academy of Sciences. Her major research areas are parallel computing, data compression and decompression, and the processing of spatial data.

Ting Wang, achieved her Ph. D. degree of Computing Mathematics in Shandong University, Shandong, China. Now she is Associate professor in Institute of Software Chinese Academy of Sciences. She is interested in high performance computing, parallel software and big data. E-mail: wangting@iscas.ac.cn