2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Removing Sequential Bottlenecks in Analysis of Next-Generation Sequencing Data

Yi Wang
Computer Science and Engineering
The Ohio State University
Columbus, U.S.A
wayi@cse.ohio-state.edu

Gagan Agrawal
Computer Science and Engineering
The Ohio State University
Columbus, U.S.A
agrawal@cse.ohio-state.edu

Gulcin Ozer
Biomedical Informatics
The Ohio State University
Columbus, U.S.A
gulcin.ozer@osumc.edu

Kun Huang
Biomedical Informatics
The Ohio State University
Columbus, U.S.A
kun.huang@osumc.edu

***Abstract*—Throughput from sequencing instruments has been increasing in an unprecedented speed, leading to an explosion of the next-generation sequencing (NGS) data, and challenges in storing, managing, and analyzing these datasets. Parallelism is the key in handling large-scale data, and some progress has been made in parallelizing important steps, like sequence alignment. However, other major steps continue to be sequential, limiting the ability to handle massive datasets. In this paper, we focus on parallelizing algorithms from two areas. The first is efficient data format conversion among a wide variety of sequence data formats, which is important for cross-utilization of different analysis modules. The second is statistical analysis. Our parallelization sequence data format converter allows sequence datasets in BAM/SAM format to be converted into multiple formats, including SAM/BAM, BED, FASTA, FASTQ, BEDGRAPH, JSON, and YAML, using both shared memory and distributed memory parallelism. The converter currently comprises three instances: SAM format converter, BAM format converter and preprocessing-optimized SAM format converter. Additionally, our converter can also support partial format conversion, to perform format conversion only on a specified chromosome region. The statistical analysis module includes parallelized non-local means (NL-means) algorithm and false discovery rate (FDR) computation. Through extensive evaluation, we demonstrate high scalability of our framework.**

***Keywords*-Next-Generation Sequencing, Data Format Conversion, Statistical Analysis, Parallelization**

I. Introduction

Throughput from sequencing instruments has been increasing at an unprecedented speed, leading to an explosion of the next-generation sequencing (NGS) data, with associated challenges in storing, managing, and analyzing these datasets. Parallelism is the key in achieving acceptable turnaround times in analysis of such data. Moreover, we cannot reduce the overall latency unless all (sequential) bottlenecks in the analysis pipeline are removed. While there has been considerable work on parallelizing the *sequence analysis* step [24], [35], [21], [11], [26], [7], [18], other steps in the analysis pipeline stay sequential, creating either a bottleneck, or reducing flexibility in cross operation of tools.

We consider the following example. Currently, there are many sequence aligners[1] available to the genomics community, but most aligners output the alignments in their own specific data formats. Unless format conversion can be performed efficiently, downstream tools may not be exchanged between different aligners. Among these sequence data formats, SAM/BAM is the most popular format, which is generated by many alignment programs[2]. To the best of our knowledge, there is no existing work that supports parallel sequence data format conversion for these complex tools. Instead, sequence data format converters commonly used today can only make use of a single core, making it extremely time-consuming to process large datasets, and becoming a bottleneck when other analysis steps are performed in parallel. Similarly, there are other statistical analysis steps, such as analysis of a massive histogram, and it is becoming increasingly important to apply parallelism to this step. Both parallel algorithms and parallel computation platforms are now quite favored in many histogram analysis scenarios, such as SNP discovery [20], BLAST [11] and GSEA [11].

We have developed a scalable sequence data analysis framework, which consists of two components, a *sequence data format converter* and a *statistical analysis module*. On one hand, the sequence data format converter allows sequence datasets in BAM/SAM format to be converted into multiple formats including SAM/BAM, BED, FASTA, FASTQ, BEDGRAPH, JSON, and YAML, using both shared memory and distributed parallelism. This converter comprises three converter instances: *SAM format converter*, *BAM format converter* and *preprocessing-optimized SAM format converter*. One of the advantages of our framework is to utilize indexing, and thus, our framework can also support *partial format conversion*, to perform format conversion only on a specified chromosome region. For many analysis phases, we avoid unnecessary computation and I/O overheads caused by blindly converting the entire datasets. On the other hand, the statistical analysis module parallelizes both the non-local means (NL-means) algorithm and the false discovery rate (FDR) computation [14].

Through extensive evaluation, we demonstrated high scalability of our framework. The system is capable of efficiently converting sequence datasets in BAM/SAM format into varieties of other sequence formats in parallel. Additionally, we also compared the sequential performance of our system against Picard [4], a widely used toolkit for manipulating

[1]http://en.wikipedia.org/wiki/List_of_sequence_alignment_software

[2]http://samtools.sourceforge.net/swlist.shtml

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.64

SAM/BAM data. Finally, the statistical analysis steps have also been parallelized effectively.

II. Background

This section provides background information on next-generation sequencing technologies, and various data formats popularly used in their context.

A. Next-Generation Sequencing Technologies

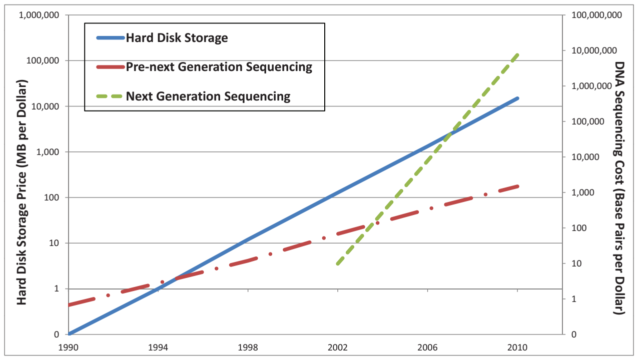


Figure 1. Historical Trends in Storage Prices versus DNA Sequencing Costs [34]

For much of the past two decades, the genome analysis industry had been dominated by *automated Sanger sequencing* [27], [16], which is also referred to as the *first-generation* technology. Despite many technical improvements during this era, the limitations of automated Sanger sequencing were quite evident. Addressing these challenges lead to the advent of next-generation sequencing (NGS) technologies. These technologies, which comprise template preparation, sequencing and visualization, and genome alignment/assembly methods, aim to present a comprehensive image of normal human genome variation.

The major advantage of NGS is the cheap and fast generation of massive amounts of sequence data, (e.g., even over one billion short reads per instrument run in some cases). Thus, one of the major challenges posed by NGS technologies is of large-scale data analysis, data integration, data storage and data movement for the massive amounts of sequence data produced by NGS instruments. As reported by others [34], [6], NGS has radically changed the *cost curve*, as shown in Figure 1 (which is reprinted from [34]). The cost of sequencing a base has fallen much faster than the cost of storing a byte, making management and analysis of sequencing data a 'big data' challenge.

B. Next-Generation Sequencing Data Formats

For efficient storage, alignment, visualization and analysis on large-scale next-generation sequencing data, a number of sequence data formats have been proposed, including SAM (Sequence Alignment/Map), BAM (Binary Alignment/Map), BED (Browser Extensible Data), FASTA, FASTQ, WIG (wiggle) and GFF (Gene Finding Feature). The following documentation [3] lists most of the major sequence data formats. Many sequencing applications can only accept certain formats, leading to the need for very efficient data format conversion. We review these formats, focusing on the most popularly used ones, which are SAM and BAM.

1) SAM: SAM (Sequence Alignment/Map) [23], [5] format is currently the de-facto standard for storing large nucleotide sequence alignments. Nowadays, most SAM datasets are produced from sequence aligners, which read sequences stored in a format called FASTQ and assign the sequences to a position compared with a known reference genome. It is also expected that SAM will be used for archiving unaligned sequence data, which is output directly from sequencers, in the near future.

SAM is a text format, which is comprised of two sections, comment lines and alignment lines. Comment lines, where each starts with an '@', may be contained in the optional SAM header. Alignment lines, which are delimited by *line breaker*, store sequence data in a series of tab-delimited ASCII columns. Each alignment line corresponds to an alignment record, and it consists of 11 mandatory fields as well as a variable number of optional fields.

2) BAM: BAM [5] is the compressed, indexable, binary form of the SAM format, which contains the same but more compact nucleotide sequence alignment information. There are two major advantages of BAM over other text alignment formats. The first advantage is its compact nature. BAM is compressed in the BGZF format, which is a block compression technique implemented on top of the standard gzip file format. All integers in BAM are little-endian, regardless of the machine endianness. Therefore, the usage of BAM can substantially reduce I/O throughput and data transfer cost during processing. The second advantage is its indexability, which leads to a fast retrieval of alignments within a particular region, by scanning only a portion of the entire alignments. The indexing is implemented by the UCSC binning scheme [17], which is a representation of the R-tree. The corresponding indexing file is in the BAI (BAM Index) format.

3) Other Sequence Formats: There are also a number of other popular sequence data formats, such as BED, FASTA, FASTQ, BEDGRAPH. The BED (Browser Extensible Data) format, which was developed by UCSC, is a tab-delimited text format for displaying transcript structures in the genome browser. FASTA format is a text-based format for representing either nucleotide sequences or peptide sequences, in which nucleotides or amino acids are represented using single-letter codes. FASTQ is another text-based format for storing both a biological sequence and its corresponding Phred quality. Its purpose is to bundle a FASTA sequence and its quality data. BEDGRAPH format is typically used to visualize the genome-wide 'scores', and it allows display of the same data value within a continuous region in a concise track format.

[3]http://www.broadinstitute.org/igv/FileFormats

III. Sequence Data Format Converter Design

In this section, we discuss the design and implementation of the parallel and scalable sequence data format converter, which includes three different converter instances for converting SAM/BAM format into other formats.

A. SAM Format Converter

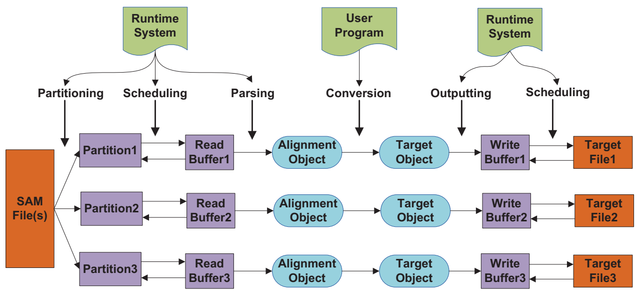


Figure 2. SAM Format Converter Execution Overview

```
Algorithm 1: partition($start, end, length, rank, N$)
 1: initialize $start$, $end$, and $length$ by evenly distributing the
    datasets to $N$ processors
 2: {adjust starting points forward, for the last $N$ - 1 processors}
 3: if $rank \neq 0$ then
 4:    allocate a temporary buffer $buf$ to read beginning data
       from the partition starting point
 5:    $index \leftarrow 0$ {detect the first line breaker}
 6:    while $buf_{index} \neq$ line breaker do
 7:       $index \leftarrow index + 1$
 8:    end while
 9:    $start \leftarrow start + index + 1$ {update start values}
10: end if
11: {assign the start value of $processor_{i+1}$ to the end value of
    $processor_i$}
12: if $rank \neq N - 1$ then
13:    $end_{rank} \leftarrow start_{rank+1}$
14:    $end \leftarrow end - 1$ {update end values}
15: end if
16: set a global barrier to wait until all the processors have
    finished the above steps
17: $length \leftarrow end - start + 1$ {update length values}
```

Our SAM format converter consists of two components: a *runtime system* and a *user program*. The runtime system is responsible for partitioning the input SAM dataset, loading partitioned data into read buffers, parsing SAM records into *alignment objects*, and writing alignment objects to the disk. The user program is designed for converting every *alignment object* into a *target object*, such as a BED text line or a FASTQ text line.

Figure 2 gives an overview of the execution flow of the SAM format converter. First, the input SAM dataset is evenly partitioned to each processor in the distributed environment. As a result, the sizes of all the partitions are exactly equal, although the number of alignments within each partition is unknown. After partitioning, the system schedules repeated loading of partitioned data into memory via the read buffer. Afterwards, a textual parsing is performed to convert each SAM text line into an alignment object, which later will be converted into a user-specified target object within the user program. Finally, each processor sends converted target objects to the write buffer and then writes to a separate target file.

Because the format conversion is entirely independent of each SAM record, there is no communication among processors after partitioning. Therefore, the most important phase for parallelization is *partitioning*. As we introduced earlier in Section II-B1, a key feature of SAM format is that, each record is delimited by a *line breaker*. Based on this feature, we propose the following partitioning strategy for the SAM format. The system first evenly distributes the dataset to each processor. In this process, it is highly likely that the initial partition boundaries are located within SAM records, rather than being located exactly at the *line breaker* positions. As a result, each initial partition is very likely to begin and end with an incomplete SAM record. Therefore, it is necessary to further adjust both the starting point and the ending point of each partition.

There are two different but equivalent implementations to perform such an adjustment of partition starting/ending point positions. The first implementation is illustrated by Algorithm 1. Here, all the processors except the first one, detect the first *line breaker* from the partition starting point. Once the first *line breaker* is found, the initial starting point is replaced by a SAM record beginning character which appears right after that *line breaker*. Afterwards, each processor sends its new partition starting point to its preceding processor for updating ending point. Finally, the partition size can be retrieved by computing the offset between updated ending point and starting point. In comparison, in the second implementation, all the processors except the last one, compute the partition ending point first, by detecting the last *line breaker* backwards. Afterwards, each processor sends its new ending point to its succeeding processor for updating starting point, and the partition size will be recomputed in the same way. Our system chooses the first implementation.

Another benefit of this converter design is its high extendibility and programmability. If the user needs to convert SAM into another format, which hasn't been supported by our system, all the user has to do is to implement a format conversion function in the user program, which converts each alignment object into the corresponding target object. All the low-level details such as parallelization, concurrency control, resource management and many other issues are abstracted within the runtime and transparent to the user.

B. BAM Format Converter

Unlike SAM format, there is no explicit delimiter between two neighboring BAM records. Therefore, the parallelization strategy used in the SAM format converter cannot be applied for the BAM format conversion process. If the input BAM datasets are initially evenly distributed, then each partition will still be wrapped with incomplete BAM records, but of unknown length, which makes the partitioned data unparsable. Thus, we realize that it is impossible to parallelize the BAM format conversion without any preprocessing.

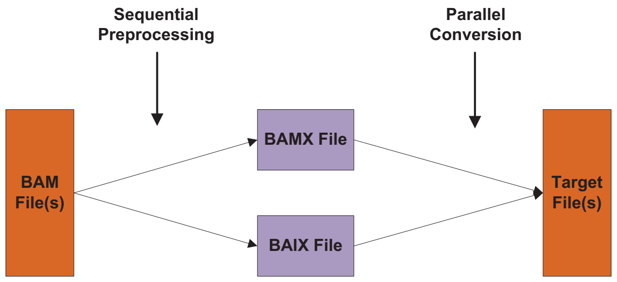


Figure 3. BAM Format Converter Execution Overview

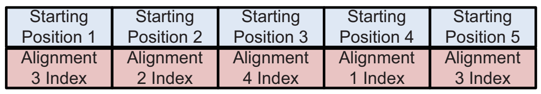
Based on this observation, the execution flow of BAM format converter, mainly consists of two phases: *sequential preprocessing* and *parallel conversion*, as also shown in Figure 3. Our BAM format parser is implemented by using the C++ API provided by BamTools [7]. Note that since this third-party library can only support reading BAM data sequentially, the preprocessing step cannot be parallelized.

During the *sequential preprocessing* phase, input BAM datasets are preprocessed to generate a *BAMX* (BAM eXtended) file as well as a corresponding *BAIX* (BAI eXtended) file. Note that both BAMX and BAIX formats are uniquely designed in our work. As the BAI file is the index file of the BAM file, the BAIX file is the index file of the BAMX file. Figure 4 shows the structure of an example BAIX file. Specifically, *starting position* is one of the mandatory fields of an alignment record, indicating where the alignment begins, and *alignment index* points to the alignment physical position in the associated BAMX file. The BAIX file stores all the alignment starting positions combined with their corresponding alignment indices within the BAMX file, in increasing order of starting position value.

The *partitioning strategy* used in the BAM format converter is mainly based on supporting the *random access* of alignment, which benefits from the *sequential preprocessing* phase. Because once the random access of alignment is allowed, the partitioning is essentially a fast retrieval of an equal number of alignments by each processor. After *partitioning*, all the remaining execution flow is as same as the SAM format converter execution flow. Furthermore, because of the generation of BAIX file, the system can support *partial conversion*, which allows format conversion on only a subset of the input BAM datasets, and thereby unnecessary computation overhead and I/O cost can be avoided.

The main purpose of *sequential preprocessing* is to facilitate the subsequent parallelization, by converting each varying-length BAM record into a regular-layout BAMX record and keeping all the records aligned. In each original BAM record, the lengths of many fields such as CIGAR data, query bases, FASTQ qualities and tag data, can significantly vary, resulting in large variance of BAM record lengths. Our preprocessing appends additional padding to all these varying-length fields so that the length of each field in the generated BAMX record is fixed. Therefore, the length of all the BAMX records is a constant value, and thus the *random access* of alignment can be easily supported.

With all the records aligned in a BAMX file, the system divides a BAMX dataset into multiple partitions, where almost every partition contains an equal number of BAMX records. During the *parallel conversion* phase, each BAMX record can be retrieved via *random access*, parsed as an alignment object, and eventually converted into a target object, in parallel.



Starting Position 1	Starting Position 2	Starting Position 3	Starting Position 4	Starting Position 5
Alignment 3 Index	Alignment 2 Index	Alignment 4 Index	Alignment 1 Index	Alignment 3 Index

Figure 4. Structure of an Example BAIX File

In practice, sometimes the user may be only interested in a data subset over a certain chromosome region, so it is unnecessary to convert the entire datasets in this case. Instead, it is more efficient to perform the format conversion over only a user-specified region. For simplicity, we call the first case as *full conversion*, and we refer to the the second case as *partial conversion*. As we stated earlier, one of the benefits of *sequential preprocessing* is that, it can support *partial conversion* based on a BAIX file. If the user specifies a sequence region with both starting position and ending position, both of these two positions can be located in the BAIX file, by using a *binary search* over the sorted alignment starting positions. Therefore, a user-specified region can be mapped to a region on the BAIX file, which we refer to as *BAIX region* for simplicity. Afterwards, the BAIX region can be easily divided into multiple equal-length subregions for each processor. Finally, each processor can retrieve all the alignments within its corresponding *BAIX subregion* via *random access* for *parallel conversion*.

The main drawback of this design is the *sequential preprocessing* cost, due to the intense I/O requirements. However, the sequential preprocessing only needs to be executed once, and this cost can be later amortized by performing parallel conversion multiple times (i.e., into different formats).

C. Preprocessing-Optimized SAM Format Converter

Our third converter instance, preprocessing-optimized SAM format converter, is developed based on another *parallelization strategy*. This parallelization strategy is a combination of the two strategies proposed earlier in Section III-A and Section III-B. We believe that a *preprocessing phase*, similar to the one in the BAM format converter execution flow, can also be used to optimize the SAM format conversion. Therefore, as Figure 5 shows, we designed a *preprocessing phase* besides a *parallel conversion phase*. However, since we can use Algorithm 1 to partition SAM datasets, this preprocessing phase can be a *parallel preprocessing* instead

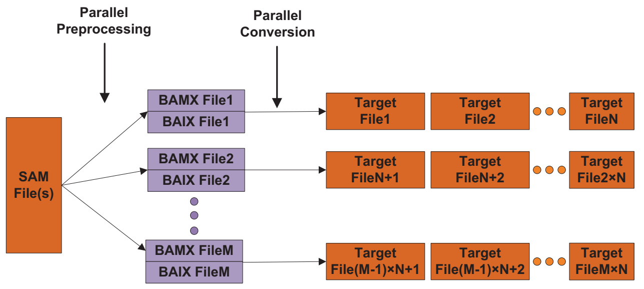


Figure 5. Preprocessing-Optimized SAM Format Converter Execution Overview

of a sequential one. After the system partitions the input SAM datasets in the preprocessing phase, each processor is responsible for converting one SAM partition into a separate BAMX file, by inserting padding and keeping all the records aligned. Additionally, the corresponding BAIX files will also be generated to support partial conversion. Note that although input SAM files are text files, the preprocessing results are compact binary files in the BAMX format, so that certain textual parsing overhead can be avoided, and the I/O cost can be reduced.

The subsequent parallel conversion phase is almost as same as the one in the BAM format converter execution flow, but the parallel conversion is only performed on a single BAMX file (and its BAIX file if needed) at a time. Therefore, if there are M processors involved in the preprocessing phase, and N processors involved in the conversion phase, there will be M BAMX files generated, and N target files generated for processing each BAMX file. In total, there will be $M \times N$ target files generated in this case.

There are three main benefits of the preprocessing-optimized SAM format converter: 1) the preprocessing cost can be reduced by parallelization; 2) the format conversion can be accelerated by the use of preprocessed BAMX files; and 3) the scalability of format conversion can be improved by the layout regularity of preprocessed BAMX files.

IV. Parallelization of Statistical Analysis Steps

By using the sequence data format converter, the user is able to convert aligned sequence data in SAM/BAM format into histogram data in BED/BEDGRAPH format in parallel. The histogram is calculated by aligning multiple sequence reads to a reference genome and accumulating the frequencies overlapped along the genome segments into binned peaks for further analysis. A quantitive field ('score') in a BED/BEDGRAPH record can be taken as such a peak value.

The computed histogram can be massive in size, and there is often a need for analysis of such histogram to make important inferences. In this section, we discuss the design of the histogram analysis module, which can perform parallel statistical analysis over the histogram data. Specifically, we parallelized both the non-local means (NL-means) algorithm and the false discovery rate (FDR) computation, proposed by Han *et al.* [14] as statistical analysis steps on histogram data.

A. NL-means Parallelization

Non-local means (NL-means) [8] algorithm was initially proposed for image denoising [38]. Recently, Zhi *et al.* have shown that NL-means algorithm is also highly effective in denoising NGS histogram data [14]. Here we briefly describe the essential formulation and the primary parameters. Formally, given a histogram with data points $V = \{v_i, i = 1, \dots, N\}$, the value of each point v_i in the histogram is updated by a weighted average of its neighborhoods in the search range R, specifically, with new value for v_i defined as

$$NL[v_i] = \sum_{j \in R} w(i,j) v_j \tag{1}$$

Here, $w(i,j)$ is the weight between v_i and v_j, and its value reflects the similarity between the two points:

$$w(i,j) = \frac{exp(-\|N(v_i) - N(v_j)\|/(2\sigma^2))}{Z(i)} \tag{2}$$

where the normalizing factor

$$Z(i) = \sum_{j \in R} exp(-\frac{\|N(v_i) - N(v_j)\|}{2\sigma^2}) \tag{3}$$

and $N(v_i)$ denotes a fixed-sized patch L centered at the point i. In practice, the NL-means algorithm requires three parameters, the search range radius r, the half patch size l, and the filtering parameter σ. The computation complexity is $\Theta(N(2r+1)(2l+1))$.

During NL-means processing, there is no communication when each point is updated, and each update operation is performed over a region of $2(r+l)+1$ length. Therefore, our parallelization strategy, which consists of three steps, has to involve replicating necessary boundary data. First, we evenly divide the 1-dimensional histogram datasets into multiple partitions $P = \{P_i, i = 1, \dots, N\}$, where N is the number of cores. Second, for each partition P_i, we replicate both a fix-sized ending region from P_{i-1} and a fixed-sized starting region from P_{i+1}, to expand the partition to P_i'. The size of each appended region is $(r+l)$, so both the starting point and ending point in the original partition P_i are able to perform NL-means processing over the enlarged partition P_i'. Finally, we perform NL-means processing over the original partition P_i, so the computation over the replicated data can be avoided.

B. FDR Computation Parallelization

The purpose of False Discovery Rate (FDR) computation is to select a threshold for region selection, based on multiple simulation datasets that are generated from random simulations, and a histogram dataset [14]. Formally, given a histogram datasets and B simulation datasets, where each dataset has M bins, the read over the i-th bin of the histogram dataset is denoted as r_i, and the read over the

i-th bin of the b-th simulation dataset is denoted as r^*_{ib}. Moreover, for the i-th bin in the histogram, the ratio that the observed data is less than simulated data is denoted as

$$p_i = \sum_{b=1}^{B} I(r_i \leq r^*_{ib}) \tag{4}$$

and the number of false peaks in the b-th round of simulation is recorded as

$$d_b = \sum_{i=1}^{M} I(\sum_{b'=1}^{B} I(r^*_{ib} \leq r^*_{ib'}) \leq p_t) \tag{5}$$

The false discovery rate for the threshold p_t is

$$FDR(p_t) = \frac{B^{-1} \sum_{b=1}^{B} d_b}{\sum_{i=1}^{M} I(p_i \leq p_t)} \tag{6}$$

The computation complexity is $\Theta(MB^2)$.

The parallelization is based on the observation that this FDR computation mainly involves two summation operations, i.e., calculating *FDR numerator* and *FDR denominator*. We can partition the datasets either in the (M-)bin direction or in the (B-)simulation direction. We choose to partition in the bin direction out of two reasons. First, in practice, the number of simulation datasets may be even less than the number of computation cores. Second, the FDR computation involves many of bin-direction comparisons, which compare the read values from the same bin location among different simulation/histogram datasets. Therefore, collecting all the read values along the same bin direction within the same partition can reduce the communication cost.

One important optimization in the parallelization is that, rather than first calculate the FDR numerator in parallel and then calculate the FDR denominator in parallel, in two separate steps, we can calculate both concurrently. In this way, we can avoid one additional global synchronization and hence improve the parallel performance. To facilitate such a parallelization, we need to take a summation permutation, which moves the bin-direction summation in the FDR numerator to the outermost, by transforming the FDR formulation as follows. We define a component of the *FDR numerator* in Equation 6 as

$$sum^{\diamond}_i = \sum_{b=1}^{B} I(\sum_{b'=1}^{B} I(r^*_{ib} \leq r^*_{ib'}) \leq p_t) \tag{7}$$

and a component of the corresponding *FDR denominator* as

$$sum^*_i = I(p_i \leq p_t) \tag{8}$$

By combining Equation 7 and 8, we can reformulate

$$FDR(p_t) = \frac{\sum_{i=1}^{M} sum^{\diamond}_i}{B \sum_{i=1}^{M} sum^*_i} \tag{9}$$

The parallel FDR computation is illustrated in Algorithm 2, After bin-direction partitioning and local summations, a master processor will collect all the local sums (i.e., $sum^{\diamond}_i$ and sum^*_i), perform a global summation to calculate both FDR numerator and FDR denominator, and finally compute the FDR value.

Algorithm 2: parallelFDR(p_{cut}, N)

```
1: evenly divide the datasets into $N$ partitions $P_i (i = 1, ..., N)$
   in the bin direction
2: compute the local sum $sum^{\diamond}_i$ for partition $P_i$
3: compute the local sum $sum^*_i$ for partition $P_i$
4: set a global barrier to wait until all the processors have
   finished the above steps
5: {compute the global sums}
6: $sum^{\diamond} \leftarrow \sum_{i=1}^{N} sum^{\diamond}_i$
7: $sum^* \leftarrow \sum_{i=1}^{N} sum^*_i$
8: $FDR(p_t) \leftarrow \frac{\sum_{i=1}^{M} sum^{\diamond}_i}{B \sum_{i=1}^{M} sum^*_i}$
```

V. Experimental Results

In this section, we evaluate the functionality and scalability of our system on a cluster of multi-core machines. We evaluate the parallel scalability of the various implementations we have described in the previous two sections. In addition, we also compare our sequential implementation system against Picard [4] which comprises Java-based command-line utilities for processing SAM/BAM files.

Our experimental dataset consists of whole genome DNA-sequencing of three mouse samples. For each sample, paired-end 90bp sequence reads were generated by Illumina HiSeq 2000 sequencing platform. Sequence reads were aligned to the mouse reference genome (mm9, July 2007, NCBI Build 37) with BWA [22]. This algorithm outputs alignment results in SAM/BAM format. Each alignment file was consisting of approximately 125 million sequences providing about 40-fold coverage of the genome.

Our experiments were conducted on a cluster of multicore machines. The system uses AMD Opteron(TM) Processor 8218 with 4 dual-core CPUs (8 cores in all). The clock frequency of each core is 2.6 GHz, and the system has an 8 GB main memory. We have used up to 256 cores (32 nodes) for our study. Our system is implemented in C++ with MPI library.

A. Sequential Comparison against Picard

Picard is a set of Java-based command-line utilities that manipulate SAM/BAM files, as well as a Java API (SAM-JDK) to help developers read and write SAM/BAM files. Picard supports format conversions between SAM/BAM and FASTQ. Therefore, in our sequential comparison experiments, we evaluated the performance of our system by converting SAM/BAM into FASTQ. The version 1.74 of Picard was used in our experiments.

We evaluated the sequential performance of our three format converter instances against Picard. Specifically, for the BAM format converter, we used both the conversion without preprocessing and the conversion with preprocessing. The experimental datasets were two datasets within a single chromosome region ‘chr1’, in the SAM and the BAM

Table I
SEQUENTIAL COMPARISON RESULTS AGAINST PICARD

Avg. Conversion Times (sec)	Our System Without Preprocessing	Our System With Preprocessing	Picard
SAM → FASTQ	3214	2804	3121
BAM → SAM	2043	1548	1425

format, respectively. The size of the SAM dataset was 37.54 GB, and the size of the corresponding BAM dataset was 7.72 GB. Specifically, we compared our implementation against Picard by converting the SAM dataset into the FASTQ format, and then we made another comparison for converting the BAM dataset into the SAM format.

Table I shows the comparison results. For conversion from SAM into FASTQ, the sequential performance of our original SAM format converter without preprocessing is slightly slower than, but quite close to, the performance of Picard. Moreover, our preprocessing-optimized SAM format converter can even outperform Picard by around 10%. This is mainly because that, the input of the preprocessing-optimized SAM format converter are binary, perfectly-aligned BAMX files instead of text files. Reading from BAMX files can save certain textual parsing overhead, and the layout regularity of BAMX files can lead to more regular data accesses and hence improve the I/O performance.

On the other hand, as for the sequential performance of converting BAM into SAM, Picard can outperform our BAM format converter without preprocessing by 30%. Picard also marginally outperforms our BAM format converter with preprocessing, although the preprocessing can also accelerate the format conversion. We believe this is because that, we used a third-party utility, BamTools, in our system to read BAM alignments. BamTools utility generates a memory object for each alignment record. To generate alignment object as the input of conversion phase in our system, an adaption from the memory object generated by BamTools to the alignment object used by our system has to be completed, leading to certain performance loss.

Recall that it was not our goal to generate fastest sequential converters. Instead, our goal was to be competitive with respect to sequential performance, which we have demonstrated.

B. Performance of SAM Format Converter

This experiment evaluates the performance of the parallelization of converting a SAM dataset into different sequence data formats, including BED, BEDGRAPH and FASTA. The experimental dataset is a subset of the entire SAM dataset, where the size is 100 GB, while the number of cores used for parallel conversion varies from 1 to 128. Figure 6 shows the results. We can see that our system is capable of scaling the performance of such a conversion over a SAM dataset. This demonstrates the effectiveness of our partitioning algorithm described in Algorithm 1, which can help achieve a good load balance. Moreover, the results also show that the conversion from SAM into BEDGRAPH

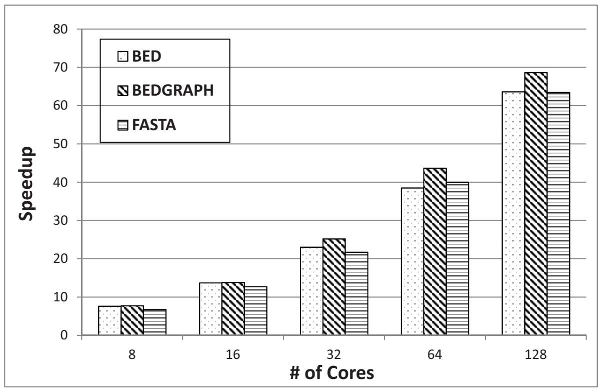


Figure 6. Conversion Speedup of SAM Format Converter

scales slightly better than the other two conversions. This is because that, as more cores are involved in the conversion, the scalability is mainly curbed by the I/O bottleneck. Since a BEDGRAPH record contains less text information than a BED or FASTA record, the conversion into BEDGRAPH is the least I/O intensive, leading to the best scalability.

C. Performance of BAM Format Converter

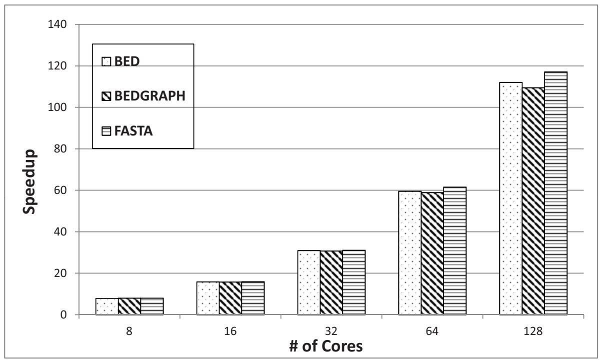


Figure 7. Full Conversion Speedup of BAM Format Converter

We next evaluate the conversion performance of our BAM format converter, by converting a BAM dataset into three different formats: BED, BEDGRAPH and FASTA. We first report the results from conversion of a *full* or entire given dataset, i.e, not any subset of the given datasets. We used a 117 GB sorted BAM dataset, and number of cores we used varies from 1 to 128. The results in Figure 7 show that the performance of the full conversion scales well as the number of cores increases. There are two reasons for such a good scalability. First, after the preprocessing phase, all the BAMX records are perfectly aligned by padding, so that the data layout has a very regular pattern. Such a regularity in layout helps improve the MPI-IO performance. Second, the conversion tasks assigned to all the processors are independent of each other.

D. Partial Conversion Performance of BAM Format Converter

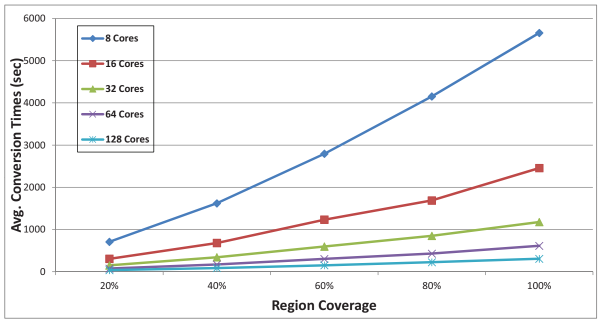


Figure 8. Partial Conversion Speedup of BAM Format Converter

We next evaluate the performance of our BAM format converter, by converting different subsets of the 117 GB BAM dataset into the SAM format. In this experiment, the subsets correspond to different chromosome regions, and are 20%, 40%, 60%, 80% and 100% of the original 117 GB BAM dataset. We scaled the number of cores from 8 to 128. The results in Figure 8 show that the conversion times for different subsets are approximately proportional to the sizes of specified regions. It demonstrates that our system can support parallel partial conversion very efficiently, because the overhead of identifying specified chromosome regions by using binary search over the BAIX file is trivial, compared with the format conversion cost.

E. Comparing Preprocessing-Optimized SAM Format Converter against Original SAM Format Converter

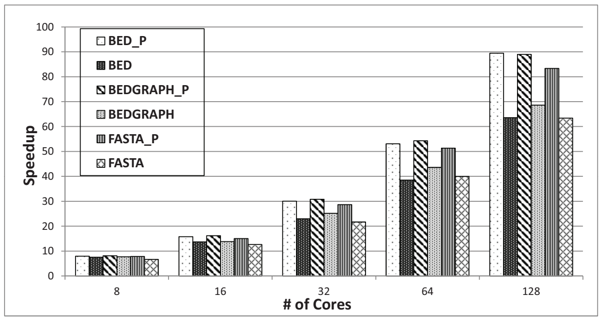


Figure 9. Conversion Speedup of Preprocessing-Optimized SAM Format Converter and Original SAM Format Converter

In this experiment, we compare the parallel performance of preprocessing-optimized SAM format converter against the original one (which does not require any preprocessing). We converted SAM format into three different sequence formats, including BED, BEDGRAPH and FASTA, by using both the preprocessing-optimized SAM format converter and the original SAM format converter, respectively. The size of the experimental SAM dataset is 15.7 GB. Figure 9 shows our experimental results. The bars end with "_P" indicate the conversion speedups of the preprocessing-optimized SAM format converter, where the preprocessing cost is excluded. In contrast, the other bars represent the conversion speedups of the original SAM format converter.

First, we can see that the scalability of the preprocessing-optimized SAM format converter is better than the original SAM format converter. This is because that each alignment within the preprocessed BAMX file(s) is perfectly aligned after preprocessing, leading to a more regular data layout. This layout regularity can help improve the MPI-IO performance. Second, we can conclude that the SAM preprocessing can accelerate the conversion. For instance, we observed that the times of converting into BED, BEDGRAPH and FASTA with the original SAM format converter on 128 cores were 16.64s, 15.10s, and 18.54s, respectively, while the corresponding times with the SAM preprocessing were 11.51s, 11.48s, and 12.80s, respectively. As a result, the performance of conversion into BED, BEDGRAPH and FASTA was improved by the SAM preprocessing by a factor of 30.8%, 24.0%, and 31.0%, respectively. This is because that the preprocessing can save certain textual parsing overhead, by storing alignments in a binary format. However, it is necessary to point out that there is a tradeoff between the conversion performance and the preprocessing cost. The more textual parsing overhead is saved during the conversion phase, the more textual parsing needs to be completed during the preprocessing phase.

F. Preprocessing Performance of Processing-Optimized SAM Format Converter

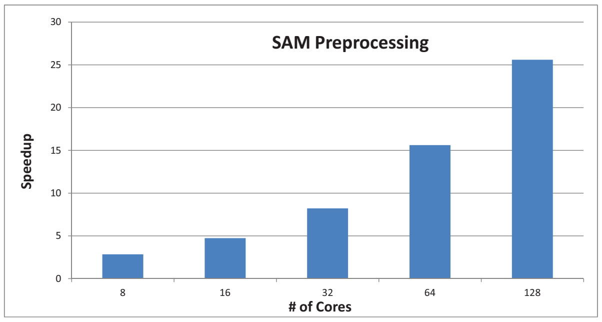


Figure 10. Preprocessing Speedup of Preprocessing-Optimized SAM Format Converter

This experiment is designed to evaluate the scalability of the SAM preprocessing step, with the same 15.7 GB SAM dataset used in Section V-E. The sequential preprocessing time was 2187s. The results in Figure 10 show that, although the scalability within a single node is mainly bridled by the I/O bottleneck, the performance scales well as the number of cores increases. It demonstrates that the SAM preprocessing can also be well parallelized in distributed environments by using the partitioning Algorithm 1.

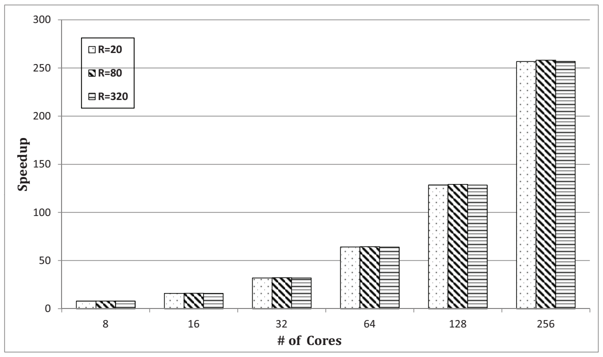


Figure 11. Speedup of NL-means Processing

G. Parallel NL-means Performance

In this experiment, we evaluate the scalability of parallel NL-means processing. We used up to 128 cores to denoise 16M bp histogram data. Recall that NL-means requires 3 salient parameters, i.e., search range radius r, half patch size l, and filtering parameter σ. Because adjusting the value of σ only affects the denoising quality rather than the processing overhead, we set a fixed value for σ. Moreover, because the value of r is usually much greater and adjusted more often than the value of l, among the 3 NL-means parameters we only varied r, from 20 to 320 bins, where the bin size was 25 bp. Particularly, for the other two fixed parameters, we set $\sigma = 10$, and $l = 15$.

NL-means processing over sequence data is quite computationally intensive, and the average sequential processing times for $r = 20$, $r = 80$ and $r = 320$ are 10213s, 41010s and 163231s, respectively. As Figure 11 illustrates, NL-means performance scales well as the number of cores grows or the search range radius increases. This is because that NL-means processing is independent of each partition, and the extra parallelization overhead, which is mainly caused by replicating a small redundant boundary data, is relatively trivial.

H. Parallel FDR Computation Performance

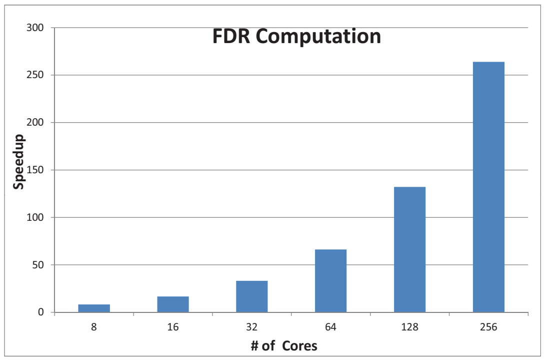


Figure 12. Speedup of FDR Computation

Our last experiment evaluates the parallel performance of FDR computation. We used up to 256 cores to process 1 histogram dataset and 80 simulation datasets, where each dataset contains 16M bins. The results in Figure 12 show that FDR computation can gain a good speedup by parallelization. The speedups compared with the sequential version that averagely consumes 1164s, are demonstrated up to 8.30, 16.60, 33.15, 66.16, 132.14, and 263.94, for 8, 16, 32, 64, 128, and 256 cores, respectively. Additionally, we can also conclude that there is certain extra speedup gained by the summation permutation in Algorithm 2.

VI. Related Work

The topic of processing next-generation sequencing data has attracted a lot of attention in the past few years.

The Genome Analysis Toolkit (GATK) [26] is designed to ease the development of efficient and robust analysis tools for next-generation DNA sequencers. GATK provides a small but rich set of data access patterns that encompass the majority of analysis tool needs. However, there is no specific utility for sequence data format conversion or the statistical analysis steps we have parallelized. Besides, currently distributed memory parallelization is not stably supported by GATK yet. Furthermore, MapReduce already has a substantial base in NGS analysis as well as other bioinformatics domains [35], [10], [29], [11], [25], [39]. Apart from GATK, a series of MapReduce-based efforts have been carried-out. This includes Cloudburst [33], Crossbow [20], Contrail [3], Jnomics [1], Myrna [19], and Bioconductor [12], which were all developed for analysis tasks such as whole genome resequencing analysis, SNP genotyping from short reads, assembly from short sequencing reads, short read alignment, and calculating differential gene expression from large RNA-seq data sets. Moreover, Biodoop [21] demonstrates that several bioinformatics applications are compatible with the MapReduce paradigm. Hadoop-BAM [28] is a novel library for the scalable manipulation of BAM datasets in Hadoop. SciMATE [36] is a novel MapReduce framework which supports various data analysis over bioinformatics data stored in HDF5 [37]. Additionally, parallel NL-means has been developed for image processing on both multicore CPUs and GPUs [32], [15], [13]. In comparison, our parallelization is applied to denoising 1-dimensional sequence data in distributed environments.

There are many software libraries for manipulating and exploring genomic datasets. SAMtools [23] is a software package that comprises various utilities for parsing and manipulating alignments in the SAM/BAM format. BamTools [7] is a fast and flexible C++ API and toolkit for manipulating and querying BAM files, including merging, filtering, and sorting them. Picard [4] supports analyzing and managing sequence data in SAM/BAM format. We have conducted extensive comparisons between our system and Picard. BEDTools [30] is another popular software suite for the comparison, manipulation, and annotation of genomic features in multiple sequence formats, including BED, SAM/BAM, GFF/GTF, and VCF. Moreover, there is a Python extension of BEDTools pybedtools [9], which provides an intuitive Python interface that extends upon much of the functionality in BEDTools. The European Molecular Biology Open Software Suite (EMBOSS) [31] is a free open source software analysis package, which aims to meet the needs of the molecular biology (e.g., EMBnet) user community. Galaxy [2] is an open, web-based platform for data intensive biomedical research. Particularly, it provides several light-weight data format converters on the web interface. However, none of these tools has provided any parallel implementation of the tasks we have focused on.

VII. Conclusions And Future Work

This paper describes implementation of a scalable sequence data analysis framework, which can help remove certain sequential bottlenecks in analysis of next-generation sequence data. To the best of our knowledge, our system is the first framework that can easily support parallel sequence format conversion in distributed environments, with three converter instances including SAM format converter, BAM format converter and preprocessing-optimized SAM format converter. Additionally, we parallelize two statistical analysis instances, which are NL-means algorithm and FDR computation.

We have extensively evaluated our implementation and compared its sequential performance against Picard. We demonstrate that our sequential performance is close to or even better than Picard. Furthermore, our system is capable of supporting a larger variety of sequence format conversions with three converter instances, scaling performance by parallelizing the conversions, and parallelizing NL-means algorithm and FDR computation. In future work, we plan to utilize certain compression techniques during the BAMX/BAIX file generation, and we also intend to apply more sophisticated indexing techniques to the BAIX structure design for supporting more partial conversion types.

Acknowledgements

We wish to thank Derek Barnett and its research group at Boston College for making BamTools open source.

References

[1] Contrail: Assembly of Large Genomes using Cloud Computing. http://contrail-bio.sourceforge.net.

[2] Galaxy. http://galaxy.psu.edu/.

[3] Jnomics. http://sourceforge.net/apps/mediawiki/jnomics/index.php.

[4] Picard. http://picard.sourceforge.net.

[5] BAM/SAM Specification, Sep 2011. v1.4-r985.

[6] M. Baker. Next-generation sequencing: adjusting to data overload. *Nature Methods*, 7(7):495–499, Jul 2010.

[7] D. W. Barnett, E. K. Garrison, A. R. Quinlan, M. Stromberg, and G. T. Marth. BamTools: a C++ API and toolkit for analyzing and managing BAM files. *Bioinformatics*, 27(12):1691–1692, 2011.

[8] A. Buades, B. Coll, and J. M. Morel. A Non-Local Algorithm for Image Denoising. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2 of *CVPR '05*, pages 60–65, Washington, DC, USA, June 2005. IEEE.

[9] R. K. Dale, B. S. Pedersen, and A. R. Quinlan. Pybedtools: a flexible Python library for manipulating genomic datasets and annotations. *Bioinformatics*, 27(24):3423–3424, Dec. 2011.

[10] J. Ekanayake, T. Gunarathne, and J. Qiu. Cloud Technologies for Bioinformatics Applications. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[11] M. Gaggero and et al. Parallelizing bioinformatics applications with MapReduce. *Cloud Computing and Its Applications*, 2008.

[12] R. C. Gentleman and et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10):R80+, 2004.

[13] B. Goossens, Q. Luong, J. Aelterman, A. Pizurica, and W. Philips. A GPU-Accelerated Real-Time NLMeans Algorithm for Denoising Color Video Sequences. In J. Blanc-Talon, D. Bone, W. Philips, D. Popescu, and P. Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, volume 6475 of *Lecture Notes in Computer Science*, pages 46–57. Springer Berlin / Heidelberg, 2010.

[14] Z. Han, L. Tian, T. Pecot, T. Huang, R. Machiraju, and K. Huang. A signal processing approach for enriched region detection in RNA polymerase II ChIP-seq data. *BMC Bioinformatics*, 13(Suppl 2):S2, 2012.

[15] K. Huang, D. Zhang, and K. Wang. Non-local means denoising algorithm accelerated by GPU. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 7497, Oct. 2009.

[16] C. A. Hutchison. DNA sequencing: bench to bedside and beyond. *Nucleic Acids Research*, 35(18):6227–6237, Sep 2007.

[17] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The Human Genome Browser at UCSC. *Genome Res*, 12(6):996–1006, June 2002.

[18] M. Kutlu and G. Agrawal. PAGE: A Framework for Easy PArallelization of GEnomic Applications. In *Proceedings of IPDPS*, 2014.

[19] B. Langmead, K. D. Hansen, and J. T. Leek. Cloud-scale RNA-sequencing differential expression analysis with Myrna. *Genome biology*, 11(8):R83+, Aug. 2010.

[20] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome biology*, 10(11):R134+, Nov. 2009.

[21] S. Leo, F. Santoni, and G. Zanetti. Biodoop: Bioinformatics on Hadoop. *Parallel Processing Workshops, International Conference on*, 0:415–422, 2009.

[22] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, July 2009.

[23] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The Sequence Alignment/Map format and SAMtools. *Bioinformatics (Oxford, England)*, 25(16):2078–2079, Aug. 2009.

[24] E. Mardis. Next-Generation DNA Sequencing Methods. *Annual Review of Genomics and Human Genetics*, 9(1):387–402, June 2008.

[25] A. Matsunaga, M. Tsugawa, and J. Fortes. CloudBLAST: Combining MapReduce and Virtualization on Distributed Resources for Bioinformatics Applications. *eScience, IEEE International Conference on*, 0:222–229, Dec. 2008.

[26] A. McKenna and et al. The Genome Analysis Toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, Sept. 2010.

[27] M. Metzker. Emerging technologies in DNA sequencing. *Genome Res*, 15(12):1767–76, dec 2005.

[28] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. Hadoop-BAM: Directly manipulating next generation sequencing data in the cloud. *Bioinformatics*, 28(6):876–877, Feb. 2012.

[29] L. Pireddu, S. Leo, and G. Zanetti. Mapreducing a genomic sequencing workflow. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 67–74, New York, NY, USA, 2011. ACM.

[30] A. R. Quinlan and I. M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2010.

[31] P. Rice, I. Longden, and A. Bleasby. EMBOSS: the European Molecular Biology Open Software Suite. *Trends Genet*, 16(6):276–7, 2000.

[32] T. Schairer, B. Huhle, P. Jenke, and W. Straber. Parallel Non-Local Denoising of Depth Maps. In *International Workshop on Local and Non-Local Approximation in Image Processing (EUSIPCO Satellite Event)*, 2008.

[33] M. C. Schatz. CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics (Oxford, England)*, 25(11):1363–1369, 2009.

[34] L. Stein. The case for cloud computing in genome informatics. *Genome Biology*, 11(5):207, 2010.

[35] R. Taylor. An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, 11(Suppl 12):S1+, 2010.

[36] Y. Wang, W. Jiang, and G. Agrawal. SciMATE: A Novel MapReduce-Like Framework for Multiple Scientific Data Formats. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 443–450. IEEE, 2012.

[37] Y. Wang, Y. Su, and G. Agrawal. Supporting a light-weight data management layer over hdf5. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 335–342. IEEE, 2013.

[38] J. Yang and Z. Fei. Broadcasting with prediction and selective forwarding in vehicular networks. *International Journal of Distributed Sensor Networks*, 2013, 2013.

[39] B. Zhang, D. T. Yehdego, K. L. Johnson, M.-Y. Leung, and M. Taufer. Enhancement of accuracy and efficiency for rna secondary structure prediction by sequence segmentation and mapreduce. *BMC Structural Biology*, 13(Suppl 1):S3, 2013.