A MapReduce Framework for DNA Sequencing Data Processing

https://doi.org/10.3991/ijes.v4i4.6537

Samy Ghoneimy, Samir Abou El-Seoud
British University in Egypt, Cairo, Egypt

***Abstract*—Genomics and Next Generation Sequencers (NGS) like Illumina Hiseq produce data in the order of 200 billion base pairs in a single one-week run for a 60x human genome coverage, which requires modern high-throughput experimental technologies that can only be tackled with high performance computing (HPC) and specialized software algorithms called "short read aligners". This paper focuses on the implementation of the DNA sequencing as a set of MapReduce programs that will accept a DNA data set as a FASTQ file and finally generate a VCF (variant call format) file, which has variants for a given DNA data set. In this paper MapReduce/Hadoop along with Burrows-Wheeler Aligner (BWA), Sequence Alignment/Map (SAM) tools, are fully utilized to provide various utilities for manipulating alignments, including sorting, merging, indexing, and generating alignments. The Map-Sort-Reduce process is designed to be suited for a Hadoop framework in which each cluster is a traditional N-node Hadoop cluster to utilize all of the Hadoop features like HDFS, program management and fault tolerance. The Map step performs multiple instances of the short read alignment algorithm (BoWTie) that run in parallel in Hadoop. The ordered list of the sequence reads are used as input tuples and the output tuples are the alignments of the short reads. In the Reduce step many parallel instances of the Short Oligonucleotide Analysis Package for SNP (SOAPsnp) algorithm run in the cluster. Input tuples are sorted alignments for a partition and the output tuples are SNP calls. Results are stored via HDFS, and then archived in SOAPsnp format. The proposed framework enables extremely fast discovering somatic mutations, inferring population genetical parameters, and performing association tests directly based on sequencing data without explicit genotyping or linkage-based imputation. It also demonstrate that this method achieves comparable accuracy to alternative methods for sequencing data processing.**

***Index Terms*—Next Generation Sequencers; short read aligners; short read alignment algorithm; DNA sequencing data processing**

I. INTRODUCTION

Today, genome sequencing machines (such as Illumina's HiSeq 4000) are able to generate thousands of gigabases of DNA and RNA sequencing data in a few hours for less than US$1,000 (a few years ago, the price was over US$100,000, and sequencing the first human genome cost about US$3 billion) [1, 2].

Success in biology and the life sciences depends on our ability to properly analyze the big data sets that are generated by these technologies, which in turn requires us to adopt advances in informatics. Map-Reduce/Hadoop and Spark enable us to compute and analyze thousands of gigabytes/petabytes of data in hours (rather than days or weeks). For example, Spark was recently used to sort 100 TB of data using 206 machines in 23 minutes.

In simple terms, DNA sequencing is the sequencing of whole genomes (such as human genomes). According to [3] "if finding DNA was the discovery of the exact substance holding our genetic makeup information, DNA sequencing is the discovery of the process that will allow us to read that information."

The main function of DNA sequencing is to find the precise order of nucleotides within a DNA molecule. Also, DNA sequencing is used to determine the order of the four bases—adenine (A), guanine (G), cytosine (C), and thymine (T)—in a strand of DNA.

The challenges of DNA sequencing are many [4], but the most important ones are:

1. There are several sequencing technologies to generate FASTQ files, and the lengths of DNA sequences are different for each sequencing technology.
2. The input data (FASTQ data) size is big (a single DNA sequence sample can be up to 900 GB).
3. With a single powerful server, it takes too long (up to 80 hours) to process one DNA sequence and extract variants such as single nucleotide polymorphisms (SNPs).
4. There are many algorithms and steps involved in DNA sequencing, so selecting the proper combinations of open source tools is a serious challenge. For example, there are quite a few mapping/alignment algorithms and parameters.
5. Scalability—that is, optimizing the number of mappers and reducers—is difficult to achieve.

A high-level DNA sequencing workflow is presented in Figure 1. As we mentioned above, our focus in this paper will be implementing DNA sequencing as a set of MapReduce programs that will accept a DNA data set as a FASTQ file and finally generate a VCF (variant call format) file, which has variants for a given DNA data set.

One of the major goals of DNA sequencing is to find variants, since most of our DNA is identical; only a very small percent differs from person to person. One important example is the identification of single nucleotide polymorphisms (SNPs). The identification and extraction of SNPs from raw genetic sequences involves many algorithms and the application of a diverse set of tools

II. THE DNA SEQUENCING PIPELINE

The DNA sequencing pipeline is illustrated in Figure 2 and includes the following key steps [6]:

1. Input data validation: performing quality control on input data such as FASTQ files
2. Alignment: mapping short reads to the reference genome
3. Recalibration: visualizing and post-processing the alignment, including base quality recalibration
4. Variant detection: executing the SNP calling procedure along with filtering SNP candidates

There is plenty of data to analyze and apply DNA sequencing to, and there are many open source algorithms for completing the previous four steps. Choice of these open source tools will significantly affect the final results.

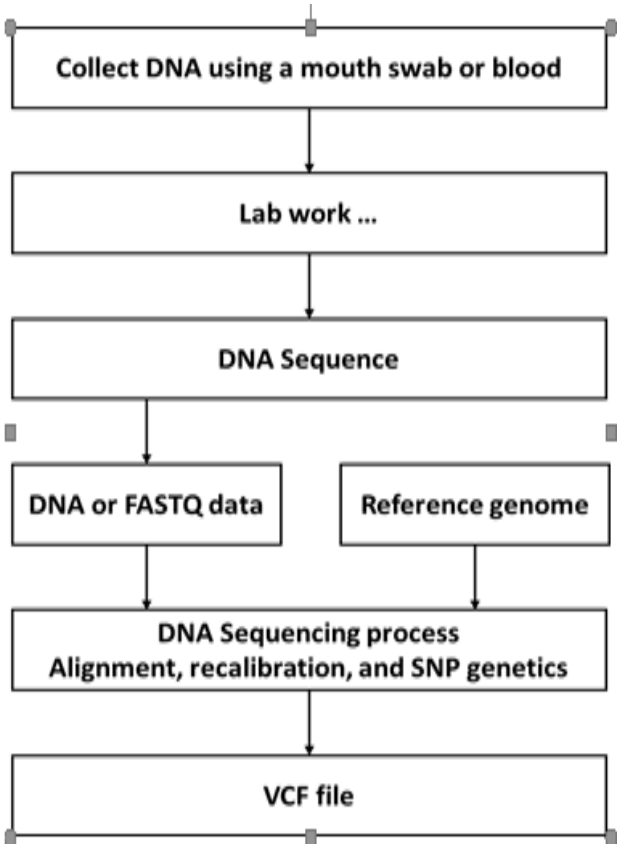

Figure 1. View of DNA sequencing

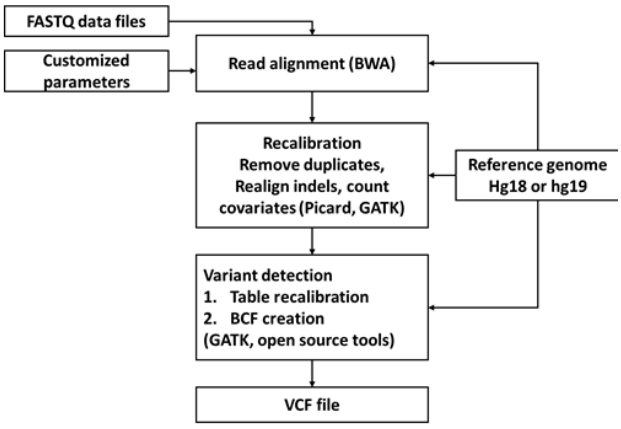

Figure 2. DNA sequencing pipeline

III. INPUT DATA FOR DNA SEQUENCING

The most common format for DNA sequencing data is FASTQ, a text-based format for storing both a biological sequence and its quality scores. For a given FASTQ file, every four lines represent a single DNA sequence.

The general syntax of a FASTQ file is as follows:

<fastq>:= <block>+

<block>:=@<seqname>\n<seq>\n[<seqname>]\n<qual> \n

<seqname>:= [A-Za-z0-9_.:-]+

<seq>:= [A-Za-z\n\.~]+

<qual>:= [!-~\n]+

A practical example is given below:

@NCYC361-11a03.q1k bases 1 to 1576

GCGTGCCCGAAAAAATGCTTTTGGAGCCGCGC
GTGAAAT...

+NCYC361-11a03.q1k bases 1 to 1576

!)))))****(((***%%((((*(((+,**(((+**+,-...

FASTQ data can be paired or non-paired. If it is paired, then our input for the DNA sequencing will be a pair of files: *left_file.fastq* and *right_file.fastq*. If it is nonpaired, there will be a single file: *file.fastq*.

IV. BACKGROUND AND RELATED WORK FOR FRAMEWORKS USED FOR DNA SEQUENCING AND MAPREDUCE

McKenna, Aaron, et al [7] have developed a framework to enable the development of analysis tools for NGS using MapReduce. This framework is called Genome Analysis Toolkit (GATK). The aim of GATK is to provide a structured programming framework that simplify the analysis of NGS by developing efficient, robust, and scale-tolerant NGS analysis tools. It is mainly based on covering most of the analysis tool needs by providing small rich set of data access patterns. GATK decouples and splits the infrastructure needed to access the NGS data, and the specific logic to each analysis tool. This way, it enables splitting some analysis calculations from data management infrastructure which provides more correctness, stability, and CPU and memory efficiency. This also results in the ability for automatic parallelization of distributed clusters and shared memory machines. GATK eliminates the problem of the complexity of analysing and manipulating the huge data generated and used while using DNA sequencing which enables the developers and researchers to focus more on the algorithms and analysis methods they are designing and developing. All of this makes GATK one of the most important programming frameworks for developing NGS tools. 1000 Genomes Project and the Cancer Genome Atlas are considered as two main examples for the developed NGS tools based on the GATK programming framework.

DePristo, Mark A., et al [8] have developed another framework which is an analytic and unified one. The aim of this framework is to explore the differences between several samples that achieve specific results while using five sequencing technologies and three experimental designs. These sequencing technologies can be summarized as following:

1. Initial read mapping
2. Local realignment around indels

3. Base quality score recalibration
4. SNP discovery and genotyping
5. Machine learning

SNP discovery and genotyping is to find all the potential variants while machine learning is for the separation between true segregating variation and machine artifacts common to NGS technologies. The developed framework provides an integrated approach to data processing and variation discovery from NGS data. This approach was designed to meet the required specifications. The experimentations done were comprehensive as several and distinct sequencing technologies were applied. The data used for the experimentation was generated from the Board Institute and 1000 Genomes project. After the experimentation of these five sequencing technologies, results showed that the new designed and developed framework achieves more accurate results than not using it. Moreover, results showed that biological DNA variant has a great impact in increasing the sensitivity and specificity of variant discovery from NGS data and that because of the following:

1. The enhanced calibration of base quality scores introduced Local realignment for indels
2. Evaluation of multiple samples from a population

The developed framework also provides more modularity and scalability that result in high quality variant and genotype calls production.

As mentioned by Schatz, Michael C., et al [9], using such parallel programmatic frameworks ease the parallel computation process to the developers and researchers with providing efficiency and fault tolerance. Another example for such frameworks is the MPI (Message Passing Interfaces) which enables the developer to design and implement parallel programs. The disadvantage of using it is its software development complication. Condor 4 is another example which is considered as a batch processing system. It is one of the most effective systems to run multiple independent parallel computations. But it is not working as efficient with the more complicated parallel algorithms. MapReduce framework is considered as one of the most efficient frameworks with most of the programs. It provides the ease to the programmer while taking care of handling job scheduling, fault tolerance, distributed aggregation and other management tasks. MapReduce framework was initially developed by Google and its main task was to streamline analyses of huge amount of webpages. However, the MapReduce implementation by Google is confidential and it is not open source, that's why Hadoop16 is more popular and works as an alternative which managed by Apache Software Foundation. The main idea of the MapReduce is to go through some Map, Reduce and shuffle parallel computational steps. These steps can be summarized in these three points:

1. Map: reads are mapped to the reference genome in parallel on multiple machines
2. Shuffle: aggregation of the alignments to be on the same chromosome and being sorted by position
3. Scan: scanning of the sorted alignments for biological events identification in each group

The feature of parallelism makes a program works better with larger clusters. Hadoop 19, 22, 25, and 26 works well with several genomic software while providing them with better scalability. However, not all genomic pipelines fits with it. In general, Hadoop and cloud computing works well with programs which have processors that work independently for long periods and don't coordinate with each other.

V. MAPREDUCE ALGORITHMS FOR DNA SEQUENCING

The first step in the DNA sequencing pipeline is validating the format of FASTQ files. With validation, we want to verify the quality of the input files. Input data validation tools enable the quality control checks on the FASTQ file format coming from high-throughput sequencing pipelines.

There are many open source tools for input data validation. For example, for FASTQ validation you have these options; FastQValidator and FastQC

In this section we will focus on mapping/alignment, recalibration, and variant detection algorithms using MapReduce.

A. DNA sequence alignment

Sequence alignment is the comparison of two or more DNA or protein sequences. The main purpose of sequence alignment is to highlight similarity between the sequences. For global sequence alignment, consider the following example with two input sequences over the same alphabet:

- Sequence 1: GCGCATGGATTGAGCGA
- Sequence 2: TGCGCCATTGATGACCA

Our output is a possible alignment of the two sequences:

- **-GCGC-ATGGAT*T*GAGC*G*A**
- ***T*GCGC*C*ATTGAT-GACC-A**

We can observe three elements in the possible alignment output:

- Perfect matches (in **bold**), Mismatches (underlined)
- Insertions and deletions (called *indels*, presented italic formatting)

For the alignment phase, we will use MapReduce/Hadoop along with the following open source tools:

- Burrows-Wheeler Aligner (BWA), an efficient program that aligns relatively short nucleotide sequences against a long reference sequence such as the human genome [5].
- Sequence Alignment/Map (SAM) tools, which provide various utilities for manipulating alignments in the SAM format, including sorting, merging, indexing, and generating alignments in a per-position format [5]. We will be working with files in the BAM format, which is the binary format of a SAM file.

Typical DNA sequencing for a single data sample (about 400–900 GB in the FASTQ file format) might take 70+ hours for a very powerful single server. The goal of the MapReduce algorithm is to find the answer in a few hours and make the solution scalable.

Since most open source tools (such as BWA, SAMtools, and GATK) for alignment, recalibration, and variant detection have Linux command-line interfaces, at each MapReduce phase our map() and reduce() functions

will call Linux shell scripts and provide the proper parameters.

To execute these shell scripts, we will use the FreeMarker templating language, which will merge Java objects and data structures with a template to create a proper shell script (see Figure 18-3). To distinguish one DNA sequence from another, for each analysis we will assign and utilize a unique GUID called the "analysis ID" (this helps to keep our input and output directories organized).

The MapReduce solution is presented in three steps, these correspond to steps 2–4 in the DNA sequencing pipeline described above. First, data is partitioned and merged during each stage.

A.1 Step 1: Alignment

Before the alignment step starts, we partition our DNA sequence FASTQ file(s) into sets of 8 million lines (or 2 million sequences; remember, in FASTQ format, each group of four lines represents a single DNA sequence). As mentioned previously, if FASTQ data is paired, our input is a pair of files: *left_file.fastq* and *right_file.fastq*. If it is non-paired, our input will be a single file: *file.fastq*. For paired data, we will partition as follows (for paired data, an alignment map() function will process *left_file.fastq.NNNN* and *right_file.fastq.NNNN* together):

left_file.fastq.0000 right_file.fastq.0000

left_file.fastq.0001 right_file.fastq.0001

left_file.fastq.0002 right_file.fastq.0002 ….

For non-paired data, we will partition as follows (for non-paired data, an alignment map ( ) function will process *file.fastq.NNNN*):

file.fastq.0000

file.fastq.0001

file.fastq.0002 ….

Each partition (also known as a chunk) will be consumed by a map( ) function. The partition size used here is (8 million lines, equal to 2 million sequences) is just for illustration purposes; the size we use for partitions should be determined by the size of your Hadoop cluster. That is, if we have a cluster of 50 nodes and each node can handle 4 mappers, then you should split your FASTQ file into 200 partitions.

For example, if the total input size is about 400 GB, then we should partition the input into 2 GB chunks (this way, we will maximize the usage of all your mappers).

The map( ) function will read the input file (one single chunk) and will generate an aligned file in BAM format. Here, the map( ) function uses BWA to perform the alignment process. Once the alignment is done, then it will extract all chromosomes (1, 2, ..., 22, 232) and save them in the MapReduce filesystem (HDFS, for Hadoop).

For example, if we have 800 partitions, we will have generated 800 files per chromosome (23 * 800 = 18,400 files). There will be only 23 reducers (one per chromosome).

The reducer will concatenate (merge and sort) all chromosomes for a specific chromosome ID. All chromosomes 1 will be concatenated into a single file called *chr1.bam*, all chromosomes 2 will be concatenated into a single file called *chr2.bam*, and so on. Then each reducer will partition the merged BAM file into small files that will be used as input to the recalibration phase.

A.1.1 Mapper for the Alignment phase

For the alignment mapper shown in Example 18-1, our solution will accept files in the FASTQ format as input and will generate partitioned chromosomes (chr1, chr2, ….chr22, chr23).

Alignment phase: map( ) function

```
1 * @param key is a key generated by MapReduce
    framework
2 * @param value is a partitioned FASTQ file (may be
    8M lines = 2M sequences)  */
3 map (key, value) {
4 // note: chr23 = concat(chrX, chrY, chrM)
5 alignedBAMFile = alignByBWA(value);
6 (chr1File, chr2File, ..., ch23File) =
    partitionByChromosome(alignedFile);
7 for (i=1, i < 24; i++) {emit(chr<i>, chr<i>File);}}
```

The alignByBWA() function accepts a partitioned FASTQ file, performs the alignment, and finally partitions the aligned file by chromosome. All of these actions are done by a shell script template. Portions of this template are listed in the following Example.

Alignment phase: nonpaired input

```
1 #!/bin/bash
2 export BWA=<bwa-install-dir>/bwa
3 export SAMTOOLS=<samtools-install-dir>/samtools
4 export BCFTOOLS=<bcftools-install-dir>/bcftools
5 export VCFUTILS=<bcftools-install-dir>/vcfutils.pl
6 export HADOOP_HOME=<hadoop-install-dir>
7 export HADOOP_CONF_DIR=<hadoop-install-dir>
    /conf
8 # data directories
9 export TMP_HOME=<root-tmp-dir>/tmp
10 export BWA_INDEXES=<root-index-dir>/ref/bwa
11 # define ref. genome
12 export REF=<root-reference-dir>/hg19.fasta
14 ### step 1: alignment
15 # the KEY uniquely identifies the input file
16 KEY={key}
17 # input_file
18 export INPUT_FILE=${input_file}
19 export ANALYSIS_ID=${analysis_id}
20 NUM_THREAD=3
21 cd $TMP_HOME
22 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE >
    out.sai
23 $BWA samse -r $REF out.sai $INPUT_FILE|
    $SAMTOOLS view -Su -F 4 - | \
24 $SAMTOOLS sort - aln.flt
25 # start indexing aln.flt.bam file
26 $SAMTOOLS index aln.flt.bam
27 # partition aligned data
28 for i in {1..22}
29 do
30 CHR=chr$i
31 $SAMTOOLS view -b -o $CHR.bam aln.flt.bam
    $CHR
32 output_file=/genome/dnaseq/output/$ANALYSIS_ID/
     $CHR/$KEY.$CHR.bam
```

```
33 $HADOOP_HOME/bin/hadoop fs -put $CHR.bam
    $output_file
33 done
34 # do the same thing for X, Y and M chromosomes
35 $SAMTOOLS view -b -o chr23.bam aln.flt.bam chrX
    chrY chrM
36 output_file=/genome/dnaseq/output/$ANALYSIS_
    ID/chr23/$KEY.chr23.bam
37 $HADOOP_HOME/bin/hadoop fs -put chr23.bam
    $output_file
38 exit 0
```

The provided shell script handles non-paired data only. If the input files are paired, then the preceding lines 25–27 are replaced by the following code in.

Alignment phase: paired input

```
1 $BWA aln -t $NUM_THREAD $RE $INPUT_
    FILE_1 > out1.sai
2 $BWA aln -t $NUM_THREAD $REF $INPUT_
    FILE_2 > out2.sai
3 $BWA sampe -r $INFO_RG $REF out1.sai out2.sai
    $INPUT_FILE_1 $INPUT_FILE_2 | \
4 SAMTOOLS view -Su -F 4 - | $SAMTOOLS sort
    - aln.flt
```

4.1.1.2 Reducer for the alignment phase

For the alignment phase, there will be exactly 23 reducers (one reducer per chromosome).

The reducer key will be a composite key of <chrID><;><analysisID>, where the chromosome ID is labeled {01, 02, 03, ..., 23}. Note the chromosome ID of 23 includes chrM, chrX, and chrY. Each reducer will merge all aligned .bam files into a single merged chr<i>.bam file:

chr<i>.bam = merge the following files:

chr<i>.bam.0000

chr<i>.bam.0001 ….

chr<i>.bam.0437 ….

After merging all the files into a single chr<i>.bam file, we partition chr<i>.bam into many small .bam files to be fed to the recalibration mapper of step 2. The partitioned files will be: chr<i>.bam.j (j = 1, 2, 3, ..., 100+)

The reduce ( ) function for the alignment phase is presented as follow.

Alignment phase: reduce() function

```
1 * @param key is a <chrID><;><analysis_id>
2 * where chrID is in (1, 2, 3, ..., 23)
3 * @param value is ignored (not used)
4 reduce(key, value) {
5 DNASeq.mergeAllChromosomesAndPartition(key); }
```

The bulk of the work lies with the DNASeq.mergeAllChromosomesAndPartition( ) method, which merges all aligned *.bam* files for a specific chromosome as shown in the following example. As mentioned previously, the final merged file is then partitioned for further processing by the recalibration phase (step 2).

mergeAllChromosomesAndPartition() method

```
1 * reducerKey=<chrID>;<analysis_id>
2 * where chrID=1, 2, ..., 22, 23 (23 includes chrM,
    chrX, chrY)  */
3 public static void mergeAllChromosomes
                    AndPartition(String reducerKey)
4 throws Exception {
5 // split the line: each line has two fields (fields are
    separated by ";")
6 String[ ] tokens = reducerKey.split(";");
7 String chrID = tokens[0];
8 String analysisID = tokens[1];
9 Map<String, String> templateMap =
                        new HashMap<String, String>();
10 templateMap.put("chr_id", chrID);
11 templateMap.put("analysis_id", analysisID);
12 mergeAllChromosomesBamFiles(templateMap);
13 partitionSingleChromosomeBam(templateMap);}
```

As we can see from the mergeAllChromosomesAndPartition( ) method, both of the helper methods, mergeAllChromosomesBamFiles( ) and partitionSingleChromosomeBam( ), use the FreeMarker template engine to pass the required Java objects and then execute shell scripts on behalf of the reducers. The definition of the mergeAll ChromosomesBamFiles( ) method is given in the following Example.

mergeAllChromosomesBamFiles( ) method

```
1 * This method will merge the following files and create
    a single chr<i>.bam file
2 * where i is in {1, 2, ..., 23}:
3 * HDFS: /.../chr<i>/chr<i>.bam.0000
4 * HDFS: /.../chr<i>/chr<i>.bam.0001
5 * ..., 6 * HDFS: /.../chr<i>/chr<i>.bam.0437
7 * Then merge all these (.0000, .0001, ..., .0437) files
     and save the result in
8 * /data/tmp/<analysis_id>/chr<i>/chr<i>.bam
9 * Once chr<i>.bam is created, then we partition it into
     small .bam files,
10 * which will be fed to RecalibrationDriver (step 2 of
     DNA sequencing)    */
11 public static void mergeAllChromosomesBamFiles
   (Map<String, String> templateMap)
12 throws Exception {
13 TemplateEngine.initTemplatEngine( );
14 String templateFileName = <freemarker-template-file-
    as-a-bash-script>;
15 // create the actual script from a template file
16 String chrID = templateMap.get("chr_id");
17 String analysisID = templateMap.get("analysis_id");
18 String scriptFileName = createScriptFileName(chrID
   , analysisID);
```

```
19 String logFileName = createLogFileName(chrID,
                                analysisID);
20 File scriptFile = TemplateEngine.createDynamic
                            ContentAsFile(templateFileName,
21 templateMap, scriptFileName);
22 if (scriptFile != null) {
23 ShellScriptUtil.callProcess(scriptFileName,
                                        logFileName);}}
```

The TemplateEngine.createDynamicContentAsFile( ) method does the magic: it takes two inputs (templateFileName and templateMap) and produces a scriptFile Name. Basically, all parameters are passed to templateFileName and then a new shell script is generated as a scriptFileName, which is then executed on a reducer's behalf.

There are two important classes, ShellScriptUtil and TemplateEngine, that merit some discussion. The ShellScriptUtil.callProcess() method accepts a shell script file (first parameter), which it executes. It then writes all logs from the script execution to a logfile (second parameter). Logging is asynchronous, meaning that as you execute the script, the logfile immediately becomes available.

The TemplateEngine class is defined in the following Example. It just implements the basic notion of a templating engine: it accepts a template (as a text file with key holders) and key-value pairs as a Java map and then creates a brand new file in which all keys in the template are replaced by values.

TemplateEngine class

```
1 import java.io.File;
2 import java.io.Writer;
3 import java.io.FileWriter;
4 import java.util.Map;
5 import java.util.concurrent.atomic.AtomicBoolean;
6 import freemarker.template.Template;
7 import freemarker.template.Configuration;
8 import freemarker.template.DefaultObjectWrapper;
9 public class TemplateEngine {
10 private static Configuration
   TEMPLATE_CONFIGURATION = null;
11 private static AtomicBoolean initialized =
                        new AtomicBoolean(false);
12 private static String TEMPLATE_DIRECTORY =
                                "/home/dnaseq/template";
13 public static void init() throws Exception {
14 if (initialized.get()) {
15 return; }
16  initConfiguration ( );
17 initialized.compareAndSet(false, true); }
18 static {
19 if (!initialized.get( )) {
20 try { init( ); }
21          catch(Exception e) {theLogger.error("TemplateEngine init failed at initialization.", e);}}}
22 private static void initConfiguration ( )
   Throws Exception TEMPLATE_CONFIGURATION
   = new Configuration( );
23 TEMPLATE_CONFIGURATION.setDirectoryFor
                                TemplateLoading (
24 new File(TEMPLATE_DIRECTORY));
25 TEMPLATE_CONFIGURATION.setObjectWrapper
                        (new DefaultObjectWrapper());
26 TEMPLATE_CONFIGURATION.setWhitespace
                                Stripping(true);
TEMPLATE_CONFIGURATION.setClassicCompatible(
true); }
61 public static File createDynamic ContentAsFile (
                                        ...){...}
```

The most important method of the TemplateEngine class, createDynamicContentAs File( ), is defined in the following Example. This method accepts a template file that has key holders and a set of key-value pairs and then generates a new file by substituting the given keys in the key holders.

TemplateEngine.createDynamicContentAsFile() method ** @param templateFile is a template filename such as script.sh.template*

** @param keyValuePairs is a set of (K,V) pairs*

** @param outputFileName is a generated filename from templateFile*

```
public static File createDynamicContentAsFile(String
templateFile,
Map<String,String> keyValuePairs,
String outputFileName)
throws Exception {
if ((templateFile == null) || (templateFile.length() == 0))
{  return null; }
Writer writer = null;  try {
 Template template =
TEMPLATE_CONFIGURATION.getTemplate(template
File);
File outputFile = new File(outputFileName);
writer = new BufferedWriter (new FileWriter
                                (outputFile) );
template.process (keyValuePairs, writer);
writer.flush ( );  return outputFile;}
finally {if (writer != null) {writer.close();}
```

A.2 Step 2: Recalibration

Recalibration is the second phase of our MapReduce DNA sequencing pipeline. In the recalibration step, each map( ) function will work on a specific aligned chromosome.

The mapper will perform duplicate marking, local realignment, and recalibration. The goal of map( ) is to create a local recalibration table filled with *covariates*. These

local covariates will be merged by the single reducer to create the final single global file (recalibration table) that will be used by the map( ) function of the third and final step of DNA sequencing, variant detection.

Following the alignment phase, we create special metadata to be used by the recalibration mappers. This metadata has the following format:

<counter><;><partitioned-bam-file><;><ref_genome><;><analysis_id>

The recalibration mapper is presented in the following Example.

Recalibration phase: map() function

```
// key is MR generated, ignored here
// value is: <counter><;><partitioned-bam-file>
  <;><ref_genome><;><analysis_id>
map (key, value) {
// actual file location will be:
// /data/dnaseq/align/ANALYSIS_ID/merged.bam.
  <KEY>
Map<String, String> tokens =
DNASeq.tokenizeRecalibrationMapperInput(value);
String reducerKey = tokens.get("analysis_id");
DNASeq.recalibrationMapper(tokens);
emit(reducerKey, value);}
public static void recalibrationMapper(Map<String,
   String> templateMap)
throws Exception {
TemplateEngine.init();
String key = templateMap.get("key");
String analysisID = templateMap.get("analysis_id");
String scriptFileName = createScriptFile-
Name("recalibration_mapper", key,
analysisID);
String logFileName = createLogFile-
Name("recalibration_mapper",
key, analysisID);
File scriptFile = TemplateEngine.createDynamic
   ContentAsFile(
"recalibration_mapper.template", templateMap,
scriptFileName);
if (scriptFile != null) {
ShellScriptUtil.callProcess(scriptFileName,
   logFileName); }}
```

The recalibration mapper template is defined as follow.

Recalibration mapper template

```
#!/bin/bash
### Recalibration mapper template
 ### call snp (get variants) up to calculation of
       ...recal.table.csv
### once recal.table.csv is created, it will be saved in
       HDFS
### input file: aligned bam file (partitioned from a
       chr<i>.bam)
## input file: aligned bam file (partitioned from a
       chr<i>.bam)
## copy HDFS_BAM_FILE to LOCAL_BAM_FILE
HDFS_BAM_FILE=${hdfs_bam_file}
BAM_FILE='basename $HDFS_BAM_FILE';
$HADOOP_HOME/bin/hadoop fs –copyToLocal
   $HDFS_BAM_FILE.
#
# put 4.recal.table.csv into GLOBAL/SHARED
   directory
#
export SHARED_RECAL_DIR=
                        /dnaseq/recal/${analysis_id}
## marking duplicates
java -Xmx4g\Djava.io.tmpdir=$JAVA_IO_TMPDIR \
-jar $PICARD_JAR/MarkDuplicates.jar \
I=$BAM_FILE \
O=2.mark.out.bam \
M=2.mark.out.metrics \
AS=true
## local realignment
samtools index 2.mark.out.bam
java -Xmx4g \
-Djava.io.tmpdir=$JAVA_IO_TMPDIR \
-jar $GATK_JAR/GenomeAnalysisTK.jar \
-T IndelRealigner \
-I 2.mark.out.bam \
-o 3.realigned.out.bam \
-R $REF \
-targetIntervals $DBSNP/dbsnp_indel.intervals \
-known $DBSNP/dbsnp_indel.vcf \
--consensusDeterminationModel KNOWNS_ONLY \
-LOD 0.4
## base quality recalibration
 java -Xmx4g \
-Djava.io.tmpdir=$JAVA_IO_TMPDIR \
-jar $GATK_JAR/GenomeAnalysisTK.jar \
-T CountCovariates \
-I 3.realigned.out.bam \
-recalFile 4.recal.table.csv \
-R $REF \
-knownSites $DBSNP/dbsnp.vcf \
-cov QualityScoreCovariate \
-cov ReadGroupCovariate \
-cov PositionCovariate \
-cov DinucCovariate
# copy result to shared directory
```

```
cp -f 4.recal.table.csv $SHARED_RECAL_DIR/
   $KEY.4.recal.table.csv
## we also need to save 3.realigned.out.bam (which
## will be needed in variant_detection_
                                    mapper.sh.template)
cp -f 3.realigned.out.bam $SHARED_RECAL_DIR/
    $BAM_FILE.3.realigned.out.bam
### so we will have:
### $SHARED_RECAL_DIR/$KEY.4.recal.table.csv
        for KEY=1, 2, 3, ....
### $SHARED_RECAL_DIR/$BAM_FILE.3.
        realigned.out.bam for KEY=1, 2, 3, ....
### (will be input to variant_detection_
                                    mapper.sh.template)
```

The recalibration reducer template is defined in the following Example.

Recalibration reducer template

```
1 #!/bin/bash
3 ### Merge all -.4.recal.table.csv files (generated by
      individual .bam files)
4 ### into a single recal.table.merged.final.txt file.
5 ###
6 ### Once recal.table.merged.final.txt is created, it will
      be saved in
7 ### /dnaseq/recal/${analysis_id}/ and will be fed into
      VariantDetectionMapper.
8 # All -.4.recal.table.csv files are in $SHARED_
                                RECAL_ DIR  directory
9 export SHARED_RECAL_DIR=/dnaseq/recal/
   $ANALYSIS_ID/
100recal_files='find $SHARED_RECAL_DIR –name
   '*.4.recal.table.csv' | sort'
11 num_of_recal_files='find $SHARED_RECAL_DIR –
   name '*.4.recal.table.csv' | wc -1'
12 ### NOTE: all calculations will take place at
      $SHARED_RECAL_DIR
13 # prepare java input files
14 java_input_files=""
15 for file in $recal_files
16 do
17 echo "preparing java input file=$file"
18 java_input_files="$file $java_input_files"
19 done
20 cd $SHARED_RECAL_DIR
21 current_dir='pwd'
22 export MERGE_COVARIATES=
                          JavaMergeCovariates
23 $JAVA_HOME/bin/java -Xms4g -Xmx12g
   $MERGE_COVARIATES \
24 -i "$java_input_files" -o recal.txt.unsorted
25 # sort the file accordingly
26 /bin/sort -t, -k 2,2n -k3,3n -k4,4 recal.txt.unsorted >
    recal.txt.sorted
27 # The recal.txt.sorted file will be used by the Variant
     Detection Mapper.
```

AStep3: Variant detection

Variant detection (also known as SNP calling) is the final phase of DNA sequencing. The goal of this step is to generate variants in *VCF* (*variant call format*; developed by the 1000 Genomes Project). The map() function will use the BAM file generated by the map() function of the recalibration step, and the final single "recalibration table" file. The map() function will use open source tools (such as GATK and SAMtools) to generate partial variants (which are raw BCF—binary call format—files). The reducer will concatenate (sort and merge) the raw BCF files to generate a single VCF file.

Once the VCF file is created, it can be used by many analytical algorithms, such as allelic frequency (covered in Chapter 21), family analysis, and the Cochran-Armitage trend test.

Variant detection is the process of finding bases in the NGS (next-generation sequencing) data that differ from the reference genome, such as hg18 or hg19; these refer to the version of the human genome assembly and determine the version of the corresponding reference annotations (for details, see http://bit.ly/build_36_1_genome).

B. DNA sequence alignment

The mapper accepts a chunked "realigned *.bam*" file and performs the following transformations on it:

- Base quality recalibration
- Variant calling and filtering

The bulk of the work is done by the DNASeq.theVariantDetectionMapper( ) method, which accepts the required parameters and creates a proper shell script from a given template.

Finally, it executes the shell script. The mapper for the variant detection phase is provided in the following Example.

Variant detection phase: map( ) function

```
1 // key: ignored, not used
2 // value: <counter><;><3.realigned.out.bam.
          <key>><;><ref_genome><;><analysis_id>
3 // index < 0 > < 1 > < 2 > < 3 >
4 // value example-1: 0001;/<dir>
                    /realigned.out.bam.0001;hg19;208
5 // value example-2: 0007;/<dir>
                    /realigned.out.bam.0007;hg19;208
6 // NOTE: THERE WILL BE ONE SINGLE REDUCER
     for variant detection:
7 // the key for output of reducer will be <analysis_id>
8 map(key, value) {
9 Map<String, String> tokens = DNASeq.tokenize
                    TheVariantDetectionMapper(value);
10 String reducerKey = tokens.get("analysis_id");
```

```
11 DNASeq.theVariantDetectionMapper(tokens);
12 emit(reducerKey, reducerKey); }
```

The theVariantDetectionMapper( ) method, shown in following Example, accepts as a parameter the analysis_id (which uniquely identifies all file directories for a specific DNA sequencing run).

theVariantDetectionMapper( ) method

```
1 public static void theVariantDetectionMapper
                    (Map<String, String> template-
Map)
2 throws Exception {
3 TemplateEngine.init( );
4 // create the actual script from a template file
5 String scriptFileName = "/dnaseq/variant_detection_
   mapper_" +
6 templateMap.get("analysis_id") + "_" +
                        templateMap.get("key") +".sh";
7 String logFileName = "/dnaseq/variant_detection_
   mapper_" +
8 templateMap.get("analysis_id") + "_" + templateMap.
                                    get("key") +".log";
9 File scriptFile = TemplateEngine.createDynamic
   ContentAsFile(
10 "variant_detection_mapper.template", templateMap,
     scriptFileName);
11 if (scriptFile != null) {
14 ShellScriptUtil.callProcess(scriptFileName,logFile
     Name); }}
```

Portions of the *variant_detection_mapper.template* are provided as follow.

Variant detection mapper template

```
1 #!/bin/bash
4 # 1. perform base quality recalibration:
5 # GATK required that the BAM file extension has to be
    .bam
6 samtools index $REALIGNED_OUT_BAM_FILE
7 #
8 java -Xmx4g \
9 -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
10 -jar $GATK_JAR/GenomeAnalysisTK.jar \
11 -T TableRecalibration \
12 -I $REALIGNED_OUT_BAM_FILE \
13 -o 4.recal.out.bam \
14 -R $REF \
15 -recalFile $SHARED_RECAL_DIR/recal.table.
    merged.final.txt
17 # 2. variant calling and filtering
18 samtools mpileup -Duf $REF -q 1 4.recal.out.bam |
     bcftools view -bvg ] - > \
19 $REALIGNED_OUT_BAM_FILE.raw.bcf
```

C. DNA sequence alignment

As noted previously, there will be *only one reducer* for all mappers. This is because we will be merging values to create a single output: a VCF file. Accordingly, the reducer does only one thing: creates a VCF file.

Variant detection phase: reducer() function

```
1 // key: <analysis_id>, which identifies all data uniquely
2 // values: ignored
3 reduce(key, values) {430 | Chapter 18: DNA Sequenc-
ing
4 DNASeq.theVariantDetectionReducer(key);
5 emit(key, key);}
```

theVariantDetectionReducer( ) method

```
1 public static void theVariantDetectionReducer(String
                                        analysisID)
2 throws Exception {
3 TemplateEngine.init( );
4 Map<String, String> templateMap =
                        new HashMap<String, String>();
5 templateMap.put("key", "-");
6 templateMap.put("analysis_id", analysisID);
7 // create the actual script from a template file
8 String scriptFileName =
                    "/dnaseq/variant_detection_reducer_" +
9 templateMap.get("analysis_id") +".sh";
10 String logFileName = "/dnaseq/variant_detection_
    reducer_" +
11 templateMap.get("analysis_id") +".log";
12 File scriptFile = TemplateEngine.createDynamic
                    ContentAsFile(
13 "variant_detection_reducer.template",
14 templateMap,
15 scriptFileName);
16 if (scriptFile != null) {
17 ShellScriptUtil.callProcess(scriptFileName
                                    , logFileName); }}
```

Portions of the *variant_detection_reducer.template* are provided as follows.

Variant detection reducer template

```
1 #!/bin/bash
2 ...
3 # call snp (get variants)
```

```
4 # concatenate all $KEY.raw.bcf files
5 #
6 FINAL_BCF_FILE=$FINAL_DIR/all.raw.bcf
7 VCF_FILE=$FINAL_DIR/var.flt.vcf
8 ...
9 ##
10 ## Concatenate BCF files. The input files are required
       to be
11 ## sorted and have identical samples appearing in the
   same order.
12 ##
13 ALL_BCF_FILES='find $RECAL_DIR/ -name
   '*.raw.bcf' | sort'
14 $BCFTOOLS cat $ALL_BCF_FILES >
   $FINAL_BCF_FILE
15 #
16 # begin bcftools & create final VCF file
17 BCFTOOLS view $FINAL_BCF_FILE | $VCFUTILS
   varFilter > $VCF_FILE
```

VI. CONCLUSION

This paper provided a unified computational MapReduce/Hadoop analytical framework to ease the development of efficient and robust analysis tools for next-generation DNA sequencers using the functional programming philosophy of MapReduce to enable various analyses, particularly population genetic analyses. This paper also presented a MapReduce solution for DNA sequencing, a very important task in the genome analysis ecosystem. Typically, DNA sequencing can be done by a powerful computer in 70 hours, but this time can be decreased to minutes by a Map-Reduce solution on a cluster of 100 nodes. The propose programming framework enables developers and analysts to quickly and easily write efficient and robust NGS tools, many of which may be incorporated into large-scale sequencing projects like the 1000 Genomes Project and The Cancer Genome Atlas.

REFERENCES

[1] Ross MG, Russ C, Costello M, et al. “Characterizing and measuring bias in sequence data.” *Gen Biol*. 2013;14:R51. https://doi.org/10.1186/gb-2013-14-5-r51

[2] Liu L, Li Y, Li S, et al. “Comparison of next-generation sequencing systems.” *J Biomed Biotechnol.* 2012; 2012: 251364.

[3] http://dnasequencing.com, last visited: June 20, 2016. https://doi.org/10.1155/2012/251364

[4] Zhao, Xi, et al. “Combining Gene Signatures Improves Prediction of Breast Cancer Survival.” Oslo, Norway: Department of Genetics, Institute for Cancer Research, Oslo University Hospital, 2011.

[5] http://bio-bwa.sourceforge.net/ ,last visited: June 20, 2016.

[6] Mahmoud Parsian, “Data Algorithms,” O’Reilly Media, 2015.

[7] McKenna, Aaron, et al. "The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data." *Genome research* 20.9 (2010): 1297-1303. https://doi.org/10.1101/gr.107524.110

[8] DePristo, Mark A., et al. "A framework for variation discovery and genotyping using next-generation DNA sequencing data." *Nature genetics*43.5 (2011): 491-498 https://doi.org/10.1038/ng.806

[9] Schatz, Michael C., Ben Langmead, and Steven L. Salzberg. "Cloud computing and the DNA data race." *Nature biotechnology* 28.7 (2010): 691.

AUTHORS

Samy Ghoneimy and **Samir Abou El-Seoud** are with the Faculty of Informatics and Computer Science, British University in Egypt, Cairo, Egypt (selseoud@yahoo.com)

Submitted 23 October 2016. Published as resubmitted by the authors 24 November 2016.