Streaming Distributed DNA Sequence Alignment Using Apache Spark

Mushtaq, Hamid; Ahmed, Nauman; Al-Ars, Zaid

DOI
10.1109/BIBE.2017.00-57

Publication date
2017

Document Version
Final published version

Published in
2017 IEEE 17th International Conference on BioInformatics and BioEngineering (BIBE)

2017 IEEE 17th International Conference on Bioinformatics and Bioengineering

Streaming Distributed DNA Sequence Alignment Using Apache Spark

Hamid Mushtaq Nauman Ahmed Zaid Al-Ars
Computer Engineering Laboratory
Delft University of Technology, 2628 CD, Delft, The Netherlands
{*H.Mushtaq, N.Ahmed, Z.Al-Ars*}*@tudelft.nl*

***Abstract*—The large amount of data generated by Next-Generation Sequencing (NGS) technology, usually in the order of hundreds of gigabytes per experiment, has to be analyzed quickly to generate meaningful variant results. The first step in analyzing such data is to map those sequenced reads to their corresponding positions in the human genome. One of the most popular tools to do such sequence alignment is the Burrows-Wheeler Aligner (BWA mem). One limitation of the BWA program though is that it cannot be run on a cluster. In this paper, we propose StreamBWA, a new framework that allows the BWA mem program to run on a cluster in a distributed fashion, at the same time while the input data is being streamed into the cluster. It can process the input data directly from a compressed file, which either lies on the local file system or on a URL. Moreover, StreamBWA can start combining the output files of the distributed BWA mem tasks at the same time while these tasks are still being executed on the cluster. Empirical evaluation shows that this streaming distributed approach is approximately 2x faster than the non-streaming approach. Furthermore, our streaming distributed approach is 5x faster than other state-of-the-art solutions such as SparkBWA. The source code of StreamBWA is publicly available at https://github.com/HamidMushtaq/StreamBWA.**

I. Introduction

DNA analysis is performed to identify mutations in the DNA indicating specific susceptibilities to certain diseases. The first step of such analysis is mapping of the sequenced reads to their corresponding positions in the human genome. One very popular tool to do that is the Burrows-Wheeler Aligner (BWA mem) [Li13]. However, BWA mem can only run with multiple threads on a single node.

Existing state-of-the-art tools, such as SparkBWA [Abuin16], do allow for running BWA mem on a cluster in a distributed fashion. However, SparkBWA requires data to be available in the HDFS (Hadoop distributed file system). Since, normally the input files are given in gzip format, this requires first uncompressing the file before uploading it to the HDFS. Subsequently, this also slows down the execution of BWA itself, since data on the HDFS has to be reformatted as appropriate input to the BWA program tasks running on the cluster. Finally, the output files produced by those BWA tasks are combined separately at the end, which also requires significant time.

In this paper, we propose a new distributed framework, StreamBWA, which allows to run BWA mem on a cluster, while the input data is being streamed directly from a compressed file. This file can either be located on the master node or on a URL, such as an https or an ftp site. This eliminates the need to spend execution time separately on downloading the file and then uncompressing it. Moreover, since the master node can stream data to the data nodes, the overhead of uploading data to the HDFS can also be hidden. The master node can also start combining the output files of BWA tasks running on the data nodes, in parallel, once they are available, further reducing the overall time. Experimental results show that, compared to SparkBWA, StreamBWA is almost 5x faster for the selected datasets on a 4 (+1 master) nodes cluster.

The contributions of this paper are as follows.

- We implemented StreamBWA: A framework that runs BWA on a Spark cluster, where the input data is streamed in parallel to the data nodes executing the BWA mem tasks.
- StreamBWA is also able to combine the output files generated by the BWA tasks in a streaming fashion.
- StreamBWA improves the efficiency of running BWA tasks by eliminating the need to reformat the input data for them (unlike SparkBWA).

This paper is organized as follows. Section II discusses big data techniques, the different stages of DNA analysis pipelines and related work. Section III presents our streaming approach. Next, Section IV discusses the implementation of StreamBWA. This is followed by Section V, which presents the evaluation results and performance analysis. We finally conclude the paper with Section VI.

II. Background

In this section, first we discuss big data techniques available for executing parallel applications on scalable computational infrastructures, and then we discuss the typical DNA analysis pipelines. Lastly, we discuss the related work.

A. Big data techniques

The MapReduce [Dean08] programming model has been one of the most prevalent big data approaches used to manage data intensive computational pipelines that need to be processed on multiple compute nodes in a scalable cluster.

This model divides the computation into two phases: map and reduce. During the map phase, input data is first formatted as key-value $<K, V>$ pairs followed by performing a specific mapping function on all of these pairs, resulting in a mapping of the input to an output set of $<K, V>$ pairs. The mapping function is executed in a distributed fashion using various mapping tasks that run locally on the data present in the nodes of the cluster. The output

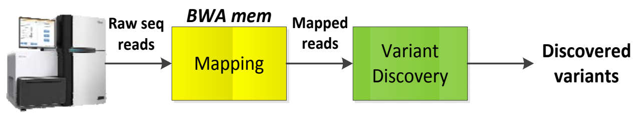

Figure 1. Typical DNA analysis and variant discovery pipeline

is then taken by the reduce tasks which first shuffle the data by grouping all the values that belong to the same key together. Afterwards, the reduce tasks compute a single output from the grouped $<K,V>$ pairs. Apache Hadoop is an example of an open source implementation of the MapReduce programming model.

One disadvantage of the MapReduce framework is that it stores all data generated between the two phases (map and reduce) on disk, and if multiple map/reduce stages are chained together, it stores the output of each stage on the HDFS. This implies significant overhead due to the intensive disk access. Another disadvantage is that the only transformations allowed are map and reduce. If a different transformation has to be applied, it has to be done by modifying the map and reduce functions, thus making development with this framework very cumbersome.

Apache Spark is a more recent big data framework that addresses the disadvantages of MapReduce listed above. First, it allows more transformations than mere map and reduce. It provides transformations such as *join*, *cogroup*, *intersection*, *distinct* and many others as predefined operations. This makes development of programs much easier as compared to the MapReduce framework. Moreover, the output of each transformation is saved in memory, but still allowing disk to be used to save data that does not fit within the available memory size. In this way, Spark avoids the overhead of disk access that is so prevalent in the MapReduce framework. In addition, Spark provides a programming interface to implement algorithms in various programming languages such as Scala, Python and Java, which allows writing programs in the language that is best suited to the problem.

B. DNA analysis pipeline

Figure 1 shows a typical DNA analysis and variant discovery pipeline. The input data set to this pipeline is the raw sequencing reads, which are obtained from a DNA sequencing machine. Since the DNA is usually over-sampled by as much as 30x to 100x by the sequencing machine, the number of such reads in a typical file could be very large, and therefore the file storing these reads is of a very large size, typically in the range of several hundreds of gigabytes for a human genome. One standard file format used today to store these reads is the FASTQ file format [Jones12].

The first step performed in the DNA analysis pipeline is DNA mapping. In this step, raw reads are mapped to a reference genome. Many classic alignment tools can be used, such as Bowtie2 [Langmead12] or the popular BWA mem. For StreamBWA, we use the BWA program unchanged, as

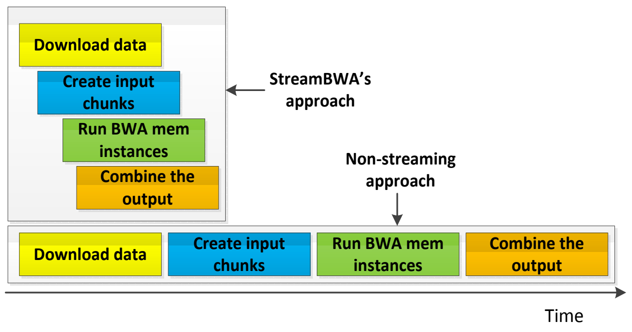

Figure 2. Timeline of streaming vs non-streaming approach

the BWA executable is executed directly by the tasks on the data nodes. This also allows us to use hardware accelerated implementations of BWA mem, such as [Ahmed15]. With StreamBWA, the output files (in SAM format) can also be combined in parallel by the master node. These files can also be combined into separate chromosomes or even load balanced chromosomal regions. The benefit of allowing such balanced output is that subsequent tools for variant discovery in the DNA analysis pipeline can efficiently work on those load balanced regions in parallel.

C. Related work

Recently, there have been a number of frameworks proposed to tackle the scalability problem of a DNA analysis pipeline. While some approaches work for the whole pipeline and use big data scalability techniques, such as SparkGA [Mushtaq17] and Halvade [Decap15], other approaches use integrated cluster scalability techniques, such as Churchill [Kelly15]. There are also scalability frameworks that focus on a single stage of the DNA analysis pipeline, such as BigBWA [Abuin12] and SparkBWA. BigBWA proposes a MapReduce-based big data solution for running BWA mem, while in SparkBWA, the same authors run BWA mem using the in-memory Apache Spark framework. In this work, we provide an Apache Spark big data scalability solution that specifically targets the DNA mapping step, similar to SparkBWA. However, our tool allows for the mapping to start while the input file is still gradually being streamed. This hides the high cost of data transfer of the big FASTQ files. Similarly, with our tool, the combination of output SAM files is also done in parallel to the the execution of DNA mapping.

III. Streaming approach

Figure 2 shows how our streaming approach can hide the latencies associated with downloading, uncompressing, distributing, and combining files. The figure indicates the timeline of our streaming approach versus the non-streaming approach. In the non-streaming approach, all these operations are performed sequentially, thereby inducing delays, whereas StreamBWA is able to perform all these operations in parallel.

Figure 3 shows the dataflow of our proposed streaming approach. In order to distribute a FASTQ file or two paired-end FASTQ files, to multiple nodes, we need to first break them down into chunks. This is done with a chunker program that we built and whose source code can be found at *https://github.com/HamidMushtaq/FastqChunker*. The FASTQ chunker program runs on the master node and can read data from either a URL (such as an ftp or an https site) or a file directly. For paired-end FASTQ files, the chunker has an option to interleave the data into single chunks as well. Moreover, the input FASTQ files can be either uncompressed or gzip compressed. The FASTQ chunker keeps on reading data from files/URLs, uncompressing it, and uploading the chunks after compressing them. The compression of the chunks can be done by using multiple threads. After the FASTQ chunker uploads a chunk, it also uploads a corresponding file to inform the StreamBWA program that the chunk has been made. As soon as a chunk has been created, a BWA task in StreamBWA, would be able to process it immediately if all resources are not already occupied by other tasks.

When a BWA task outputs a SAM file, it also uploads a corresponding flag to notify the SAM files combiner part of the StreamBWA, which has the task of combining the SAM files into one. It is also possible for StreamBWA to create one SAM file for each chromosome, or even create SAM files for chromosomal regions. While the BWA tasks are running on the Spark cluster, the SAM files combiner of StreamBWA is running in parallel.

Since the chunker is a separate program, it can also be used with other programs. For example, we successfully combined our chunker utility with the SparkGA program, so that like StreamBWA, SparkGA could also process chunks on the fly.

IV. Implementation

Our framework consists of two utilities. One is the StreamBWA tool and the other is the chunker program. The chunker reads the input gzipped file, creates compressed chunks and uploads them to the HDFS. The StreamBWA reads those chunks on the fly, and also combines the resulting SAM files into either a single combined output, into combined output files for each chromosome or into files grouped by chromosomal regions, each file representing one such chromosomal region. The chunker and StreamBWA are discussed next.

A. Chunker

The chunker utility is written in Scala and uses *Future* operations to create compressed chunks with parallel threads. In one iteration, it reads N number of blocks from a file stored either locally or at a URL, where N is the number of *Futures* (threads). After reading those N blocks, each block is given to a separate thread. This way, N parallel threads can

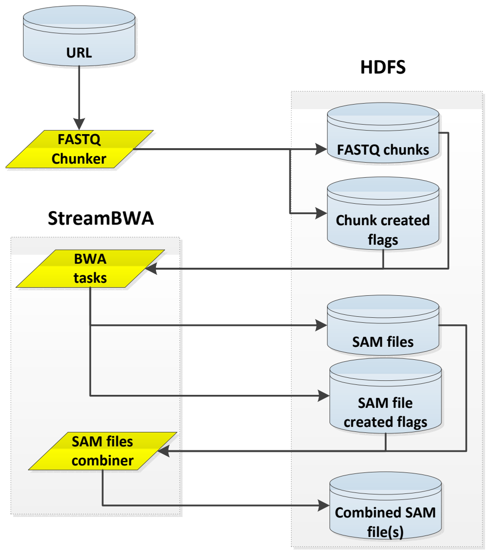

Figure 3. Dataflow of the proposed streaming approach

compress blocks and upload them to the HDFS. The Halvade upload tool is a similar utility. However, unlike the Halvade upload tool, which reads data line by line, our tool reads a block of uncompressed data at once from the gzipped file, and at the end of that block looks for the reads boundary. That is, it checks where the last read is ending, it takes the data till the last read, and puts the remaining part into a buffer, which we call the leftover buffer. This data from that leftover buffer is then appended to the data of the next block. Reading data block by block, rather than line by line, is the reason our chunking utility performs significantly better than the Halvade upload utility, as shown by the evaluation results.

After uploading a compressed chunk to the HDFS, a status file is also uploaded. This status file is just an empty file with an ID, to signal to the StreamBWA program that the corresponding chunk has been uploaded. IDs are just numbers, starting from 0. If N threads are being used, chunks from 0 to N-1 would be uploaded first, followed by chunks from N to N*2-1 and so on. This chunker program runs locally on the master node. When all the chunks have been uploaded, a sentinel file is also sent to signal that all files have been uploaded.

The algorithm for paired-end FASTQ files chunking is shown as Algorithm 1. Here, bArrArr1 and bArrArr2 are arrays of type ByteArray, which is a class we created to hold the data read in each iteration. These elements are allocated once with a big enough size, so that they can be used again and again, thus avoiding creating a new element each time. Using *new* repeatedly would increase memory and garbage collection overhead. Similarly gis1 and gis2 are objects of type GZInput which we created to read gzipped compressed data. It is different from the standard GZInputStream class, since it can read fixed number of uncompressed bytes. In

Algorithm 1: Paired FASTQ chunking

```
while !endReached do
    for i = 0 until nThreads do
        if endReached then
            bArrArr1(i).setLen(0)
            bArrArr2(i).setLen(0)
            bytesRead(i) = -1
        else
            bytesRead(i) = gis1.read(tmpBuf1)
            gis2.read(tmpBuf2)
            if bytesRead(i) == -1 then
                endReached = true
                bArrArr1(i).copyFrom(leftOver1)
                bArrArr2(i).copyFrom(leftOver2)
            else
                /* leftOver1 is null on the
                   First ever iteration      */
                if leftOver1 == null then
                    leftOver1 = new ByteArray(bufSize)
                    leftOver2 = new ByteArray(bufSize)
                    bArr1.copy(tmpBuf1, 0, bytesRead(i))
                    bArr2.copy(tmpBuf2, 0, bytesRead(i))
                else
                    bArr1.copy(leftOver1)
                    bArr1.app(tmpBuf1, 0, bytesRead(i))
                    bArr2.copy(leftOver2)
                    bArr2.app(tmpBuf2, 0, bytesRead(i))
                splitAtRB(bArr1, bArrArr1(i), leftOver1)
                splitAtRB(bArr2, bArrArr2(i), leftOver2)
        if f(i) != null then
            Await.result(f(i), Duration.Inf)
        gzOutStreams(i).synchronized {
            baFuture1(i).copy(bArrArr1(i))
            baFuture2(i).copy(bArrArr2(i))
        }
        f(i) = Future {
            gzipOutStreams(i).synchronized {
                writeToChunk(i)
            }
        }
waitForFuturesAndCloseGZOutSreams()
```

the constructor, we can specify the blocksize, and then GZInputStream would read exactly the same number of bytes (uncompressed) each time. The way the algorithm works is that in each iteration, we read blocks for each *Future*, where number of *Futures* is equal to *nThreads*. From the data read from each FASTQ file, the data is split at the last read, so that there are no incomplete reads. The last remaining read is then put into leftover buffers. The data from the leftover buffers is then appended in the beginning of the block of the next iteration. Thereafter the data read is copied into baFuture1 and baFuture2, which are buffers passed to the corresponding *Future*.

The data for a chunk is then interleaved in a *Future*, as shown in algorithm 2. When the data in the output stream becomes greater than the allowed chunk size, it is written to the chunk. Next time the data would be written to another chunk.

While future *i* is interleaving data and writing it to a chunk, the iteration *i* could be fetching data from the input gzipped paired-end FASTQ files in parallel. In this way, when the next instance of future *i* starts, its input data might already be there. With this mechanism, we are able to fully or partially hide the the time it takes to compress the chunks and upload them to the HDFS.

Algorithm 2: Writing of interleaved chunk

```
if baFuture1(i).getLen() != 0 then
    content = interleave(baFuture1(i), baFuture2(i))
    gzOutStreams(i).write(content)
    if gzOutStreams(i).getSize() > chunkSize then
        gzOutStreams(i).close()
        data = gzipOutStreams(ti).getByteArray
        writeChunkFile(chunkCtr(i) + ".fq.gz", data)
        writeStatusFile(chunkCtr(i))
        chunkCtr(i) = chunkCtr(i) + nThreads
        baos = new ByteArrayOutputStream
        gzOutStreams(i) = new GZIPOutputStream1(baos,
          compLevel)
```

Since, the chunker utility is a separate program and not a part of StreamBWA, it means it can also be used with other genomic pipeline tools. For example, we have also successfully interfaced the chunker utility with our SparkGA tool, to perform chunking in parallel on the master node.

B. StreamBWA

The StreamBWA program can be either run in client or cluster mode. In client mode, a controller part of StreamBWA (called the driver) runs on the master node, while the executors are run on the data nodes. On the other hand, in cluster mode, the driver is also run on one of the data nodes.

Each BWA task on the data node is assigned an ID. The BWA tasks are scheduled such that the task with the smallest ID, runs first. Each such task then waits for the corresponding chunk to exist in the HDFS. This can be done by looking at the corresponding status file. As soon as that status file appears, the task grabs the corresponding chunk, uncompresses it and execute BWA with it. This means that in the beginning, the first group of tasks have to wait for the chunker to have made the chunks. However, by the time the next group of tasks are scheduled, the chunker might already have placed the chunks for them, as the chunker is continuously being run in parallel on the master node. If there appears a task whose ID is greater than the total number of chunks, that task would know that it has no corresponding chunk to process, by seeing that the sentinel file has been uploaded by the chunker, while no file with that ID was produced by the chunker.

As discussed before, the StreamBWA program can also combine the output chunks into a single output file, or files grouped by chromosomes or chromosomal regions. Moreover, these combined output files can either be placed on the HDFS or on the local file system of the master node.

Table I
COMPARISON OF OUR CHUNKING UTILITY WITH HALVADE UPLOAD TOOL (CL = COMPRESSION LEVEL)

Program	**D1 chunking time (mins)**	**D1 chunks size (GB)**	**D2 chunking time (mins)**	**D2 chunks size (GB)**
Halvade upload tool (cl: 6)	53.5	74.7	20	28.4
StreamBWA (cl: 6)	39.5	74.7	13	28.4
StreamBWA (cl: 5)	33	77.5	12	29
StreamBWA (cl: 4)	32.5	79.2	12	29.5
StreamBWA (cl: 3)	32.5	80.5	12	30.4

The combiner part is implemented using Scala *Futures*, which runs on the driver part of StreamBWA. As soon as one task completes its execution of BWA, it uploads the output SAM file to the HDFS. Besides that, it also uploads a status file. The combiner keeps looking for those status files to know when to merge those output SAM files. For this, the combiner maintains a *Set* data structure, which contains the IDs of all the SAM files which have been already merged. By taking the diff of the IDs of the status files and the IDs in that *Set*, the combiner knows which files to merge. It repeats this step, until all files have been merged. To increase efficiency, more than one *Future* can be used. When a combiner has to be used, each task running BWA, outputs a compressed SAM file. By using multiple *Futures* (threads), the decompression of multiple compressed SAM files can be done in parallel.

V. EVALUATION RESULTS

We tested the results on an IBM Power7+ cluster with 4 data nodes + 1 master node. Each node has two sockets that host a Power7+ processor. In total, a node has 16 physical cores and 128GB of memory. Each Power7+ core is capable of 4-way simultaneous multithreading. Spark is run over YARN on that platform. Although the master node in our cluster also has 128GB of memory, we restricted the use of memory to less than 16 GB for the master node. This is because, in a typical scenario, the master node is much less powerful compared to the data nodes, as the master node is expected to remain idle most of the time. Since, our framework exploits this idleness of the master node, without adding extra cost to it, to emulate that situation, we kept the memory usage of the master node to less than 16GB.

We tested and compared our framework with other solutions using dataset D1 (NA12878D_HiSeqX_R1 and NA12878D_HiSeqX_R2 from [NA12878]) and dataset D2 (ERR022066_1.filt and ERR022066_2.filt from the data/HG00109/sequence_read folder of [1000genomes]). The total uncompressed size of D1 is 272GB while that of D2 is 90GB.

A. Evaluation of the chunker utility

Table I compares the performance of our chunking utility with that of the Halvade upload tool. These results are also illustrated in Figure 4. The table and figure show that for both datasets, our chunking utility outperforms the Halvade upload tool by up to 35% (at the same compression level). The reason for this increased performance is due to the decreased processing of the input data. While Halvade reads the input gzipped data line by line, our chunking utility can read blocks of data, and demarcate them properly on the read boundaries.

The results also show that by lowering the compression level from the default 6, we gain even more speedup of up to 65% at the cost of slightly bigger files. This speedup is the result of a faster gzip compression at lower compression levels, obviously. We were not able to compare with the Halvade upload tool at a lower compression level, as it only allows the default compression level.

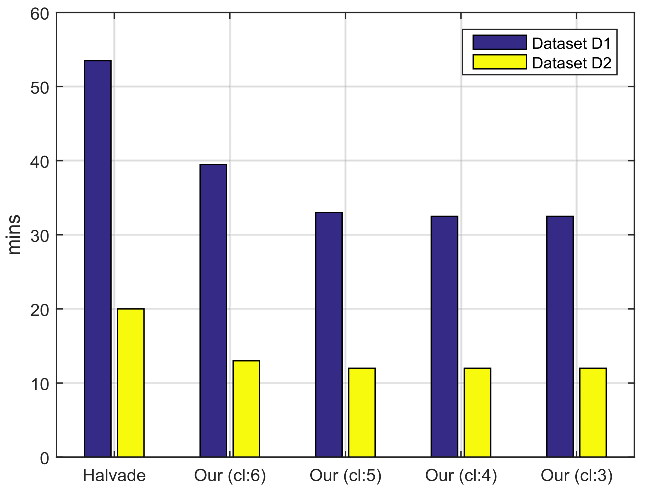

Figure 4. Performance comparison between Halvade's and our chunker utility (cl = compression level)

Table II
COMPARISON OF STREAMBWA WITH NON-PARALLEL STREAMBWA AND SPARKBWA

Process	**D1, size = 272GB (mins, % of total)**	**D2, size = 90GB (mins, % of total)**
SparkBWA		
Download	45 (7%)	12.5 (12%)
Uncompress	127 (20%)	30 (29%)
Upload to the HDFS	19 (3%)	6 (6%)
BWA	368 (59%)	33.5 (33%)
Combine	70 (11%)	21 (20%)
Total	**629**	**103**
Non-streaming SreamBWA		
Download	45 (18%)	12.5 (25%)
Chunker	32.5 (13%)	12 (24%)
BWA	104 (41%)	5 (10%)
Combine	70 (28%)	21 (42%)
Total	**251.5**	**50.5**
Streaming SreamBWA		
Chunker ‖ BWA	105 (79%)	12 (65%)
Chunker ‖ BWA ‖ Combine	120 (91%)	15.5 (84%)
Download ‖ Chunker ‖ BWA ‖ Combine	**132.5**	**20**

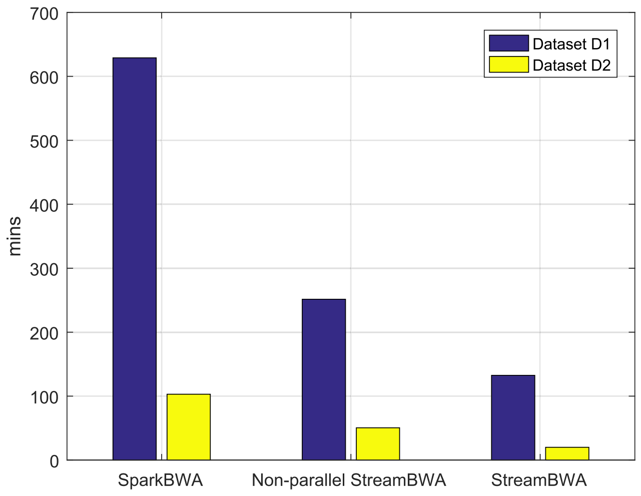

Figure 5. Comparison of StreamBWA with the non-parallel based approach and SparkBWA

B. Comparison of StreamBWA with SparkBWA

Table II shows a comparison of StreamBWA with its non-parallel counterpart and SparkBWA. The total times for each solution are shown in bold, and are plotted in Figure 5. The results show that StreamBWA is about 5x faster than SparkBWA. The main reason being that StreamBWA performs uncompression/uploading, BWA mapping and combining of output, in parallel, while SparkBWA has to do those sequentially. Furthermore, even after the data has been uploaded to the HDFS, SparkBWA has to group data into input files which are to be given to the BWA tasks. For that purpose, it has to first make RDDs by using expensive operations, such as *join* and *sortByKey*. For this reason, even the non-parallel version of StreamBWA is up to 2.5x faster than SparkBWA, as it does not require this extra step.

VI. Conclusions

In this paper, we presented StreamBWA, an open source tool (publicly available at https://github.com/HamidMushtaq/StreamBWA), which can be used for performing the Burrows-Wheeler Aligner (BWA mem) algorithm on a Spark based cluster. Not only is StreamBWA able to run BWA mem tasks in a distributed fashion on a cluster, it can also stream data to those BWA mem tasks running on the data nodes on the fly. Moreover, it also combines the output files of the BWA mem tasks into one or multiple SAM files in a streaming fashion.

Results show that this streaming distributed approach is approximately 2x faster than the non-streaming approach. In addition, since unlike other state-of-the-art approaches, like SparkBWA, which have to format data in a specific way, even after it being uploaded to the HDFS, StreamBWA formats data before uploading it to the HDFS, thus further saving time. Due to this reason, even the non-streaming version of StreamBWA is faster than SparkBWA by up to 2.5x, with the streaming version being approximately 5x faster on the selected datasets. Lastly, unlike SparkBWA, besides allowing the output to be combined into a single file, StreamBWA also allows to group output data by chromosomes or even load balanced chromosomal regions. Arranging output into such chromosomal regions is useful for optimizing the performance of the subsequent tools of a DNA analysis pipeline.

References

[Ahmed15] N. Ahmed, *et al.*, ”Heterogeneous Hardware/Software Acceleration of the BWA-MEM DNA Alignment Algorithm”, ICCAD’15, pp. 240-246, 2015.

[Auwera13] G.A. van der Auwera, *et al.*, ”From FASTQ Data to High-Confidence Variant Calls: The Genome Analysis Toolkit Best Practices Pipeline”, Current Protocols in Bioinformatics, 43:11.10.1-11.10.33, 2013.

[Decap15] D. Decap, J. Reumers, C. Herzeel, P. Costanza and J. Fostier, ”Halvade: scalable sequence analysis with MapReduce”, *Bioinformatics*, btv179v2-btv179, 2015.

[Bio] http://www.bioplanet.com/gcat

[Gusfield97] D. Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK.

[Picard] https://broadinstitute.github.io/picard

[NA12878] http://allseq.com/knowledge-bank/1000-genome/get-your-1000-genome-test-data-set

[1000genomes] ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/phase3

[Dean08] J. Dean and S. Ghemawat, ”MapReduce: Simplified Data Processing on Large Clusters”, *Commun. ACM*, vol. 51, no. 1, 2008.

[Zaharia10] M. Zaharia, *et al.*, ”Spark: cluster computing with working sets”, *HotCloud’10*, USENIX Association.

[Abuin16] J.M. Abuin, J.C. Pichel, T.F. Pena and J. Amigo, ”SparkBWA: Speeding Up the Alignment of High-Throughput DNA Sequencing Data”, *PLoS ONE 11.5*, e0155461, 2016.

[Jones12] D.C. Jones, W.L. Ruzzo, X. Peng and M.G. Katze, ”Compression of next-generation sequencing reads aided by highly efficient de novo assembly”, *Nucleic Acids Research*, 2012.

[Kelly15] B.J. Kelly, *et al.*, ”Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics”, *Genome Biology*, vol. 16, no. 6, 2015.

[Li13] H. Li, ”Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM”, arXiv:1303.3997 [q-bio.GN], 2013.

[Mushtaq15] H. Mushtaq and Z. Al-Ars, ”Cluster-based Apache Spark implementation of the GATK DNA analysis pipeline”, *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pp. 1471-1477, 2015.

[Mushtaq17] H. Mushtaq, *et al.*, ”SparkGA: A Spark Framework for Cost Effective, Fast and Accurate DNA Analysis at Scale”, *ACM Conference on Bioinformatics, Computational Biology and Health Informatics (ACM-BCB)*, pp. 148-157, 2017.

[Langmead12] B. Langmead and S.L. Salzberg, ”Fast gapped-read alignment with Bowtie 2”, Nature Methods, vol. 9, no. 4, pp. 357-359, 2012.

[Abuin12] J.M. Abun, J.C. Pichel, T.F. Pena and J. Amigo, ”BigBWA: approaching the BurrowsWheeler aligner to Big Data technologies, Bioinformatics”, vol. 31, no. 24, pp. 40034005, 2015.