Cluster-Based Apache Spark Implementation of the GATK DNA Analysis Pipeline

Hamid Mushtaq and Zaid Al-Ars
Computer Engineering Laboratory
Delft University of Technology
{H.Mushtaq, Z.Al-Ars}@tudelft.nl

Abstract—**Fast progress in next generation sequencing has dramatically increased the throughout of DNA sequencing, resulting in the availability of large DNA data sets ready for analysis. However, post-sequencing DNA analysis has become the bottleneck in using these data sets, as it requires powerful and scalable tools to perform the needed analysis. A typical analysis pipeline consists of a number of steps, not all of which can readily scale on a distributed computing infrastructure. Recently, tools like *Halvade*, a Hadoop MapReduce solution, and *Churchill*, an HPC cluster-based solution, addressed this issue of scalability in the GATK DNA analysis pipeline. In this paper, we present a framework that implements an in-memory distributed version of the GATK pipeline using Apache Spark. Our framework reduced execution time by keeping data active in the memory between the map and reduce steps. In addition, it has a dynamic load balancing algorithm that better utilizes system performance using runtime statistics of the active workload. Experiments on a 4 node cluster with 64 virtual cores show that this approach is 63% faster than a Hadoop MapReduce based solution.**

I. Introduction

Next generation DNA sequencing platforms, such as the Illumina HiSeq can generate high-throughput DNA sequencing data that require high-performance computational solutions to meet the needs of the sequencing platform. One widely used post-sequencing DNA analysis tool set is the Best Practices GATK pipeline from the Broad Institute [Auwera13]. While some stages of that pipeline, such as the Burrows-Wheeler Aligner (BWA) [Li13], are highly scalable, other stages are not, when executed on a distributed computing infrastructure.

A couple of solutions have been proposed to deal with the scalability challenge of the GATK pipeline. One example is the Churchill DNA analysis pipeline, which addresses this challenge using an input data set segmentation method to achieve parallelization across chromosomal regions [Kelly15]. The approach is able to efficiently utilize the available computational resources by combining the different analysis stages into a single tightly-integrated computational pipeline that runs on a custom-made high-performance computing (HPC) cluster. However, the use of proprietary algorithms makes this solution inaccessible and difficult to test and evaluate by third parties. On a 48-core Dell R815 server, Churchill achieves a speedup of 10x compared to a straight forward implementation of GATK with in-built multithreading enabled.

Another solution proposed to solve the scalability challenge of the GATK pipeline is the Halvade framework, which implements the different GATK pipeline stages using a Hadoop-based MapReduce approach [Decap15]. This enables standardized open-source pipelines, such as GATK, to be executed in parallel on a multi-node and/or multicore compute infrastructure in an efficient manner. Like Churchill, Halvade utilizes an input data set segmentation method to achieve scalability in all pipeline stages. One drawback of the Halvade framework though is that it is implemented as a classic Hadoop MapReduce based big data solution that is heavily disk oriented, which leads to disk access overhead due to the large data set sizes of genomics applications. In addition, the computational load of different stages of the pipeline is statically distributed across chromosomal regions, which results in non-optimal load balancing of the input data set. On a 24-core dual-socket Xeon E5-2695 server, Halvade is able to achieve a speedup of 2.5x compared to a GATK run with multithreading enabled.

This paper presents a big data framework to parallelize the different stages of the GATK pipeline using Apache Spark. The framework uses an input data set segmentation approach to achieve scalability similar to the approaches mentioned above. Our framework, however, enables scalability on standardized open-source pipelines using a dynamic load balancing algorithm that divides chromosomal regions according to the number of mapped reads to each chromosome rather than the static length of the chromosomes. This division of regions is done in-memory and therefore incurs minimal cost. We can sum up our contributions for this paper as follows.

- We implemented a cluster-based parallel framework for DNA analysis pipelines using Apache Spark.
- We perform load balancing by dividing chromosomal regions according to the number of reads mapped to each chromosome.

In Section II, we discuss the different stages of a DNA analysis pipeline. Section III presents the Apache Spark framework we implemented to enable pipeline scalability. This is followed by Section IV, which discusses the performance and accuracy evaluations. We finally conclude the paper with Section V.

II. Background

Figure 1 shows a typical DNA analysis and variant calling pipeline. The input data set to the pipeline is generated by sequencing a DNA sample in a sequencing machine and acquiring the DNA sequencing data. This is done in a massively parallel fashion with millions of short DNA pieces (called *short reads*) being sequenced at the same time. These reads are stored in large files of sizes in the range of hundreds of giga bytes. The DNA is usually over-sampled, resulting in generating multiple short reads for each segment of the DNA, typically with a coverage ranging from 30x to 80x, depending on the experiment. One standard file format used today to store these reads is called the FASTQ file format [Jones12]. Once the input data becomes available, the first step in the DNA analysis pipeline is to align the sequenced reads to a human reference

Algorithm 1 Parallelized GATK pipeline

```
1:  procedure GATKParallel
2:      ▷ Get the names of the input chunks
3:      $chunksName \leftarrow sc.parallelize(inputChunksName)$

4:      ▷ chrToSamRecord contains elements of type <chromosome number, SAM record>
5:      $chrToSamRecord \leftarrow chunksName.flatMap(x => bwaRun(x))$
6:      $chrToSamRecord.cache()$

7:      ▷ avgSamRecords = average number of SAM records per chromosome
8:      $avgSamRecords \leftarrow chrToSamRecord.count/NumberOfChromosomes$
9:      ▷ chrToNumSamRecords contains the number of SAM records for each chromosome
10:     $chrToNumSamRecs \leftarrow chrToSamRecord.map\{case(chr, SAM) => (chr, 1)\}.reduceByKey(\_ + \_)$
11:     ▷ chrInfo contains values of type <number of SAM records, chromosome length> keyed by chromosome number
12:     $chrInfo \leftarrow CreateMap(chrToNumSamRecs.map\{case(chr, nSR) => (chr, (nSR, ChrLen(chr)))\}.collect)$

13:     ▷ Load balancing
14:     ▷ chrToSamRecordBal contains elements of type <chromosome region, SAM record>
15:     $chrToSamRecordBal \leftarrow chrToSamRecord.map(x => balanceLoad(x, chrInfo, avgSamRecords))$
16:     $numOfRegions \leftarrow chrToSamRecordBal.map(x => x.\_1).reduce((x, y) => max(x, y))$

17:     ▷ Variant calling
18:     ▷ vcf contains elements of type <chromosome, output line>
19:     $chrToSamPartitioned \leftarrow chrToSamRecordBal.partitionBy(new\ CustomPartitioner(numOfRegions + 1))$
20:     $vcf = chrToSamPartitioned.mapPartitions(variantCallingWrapper)$

21:     ▷ Write results to the file
22:     $writeVCF(vcf.distinct.sortByKey.collect)$
23: end procedure
```

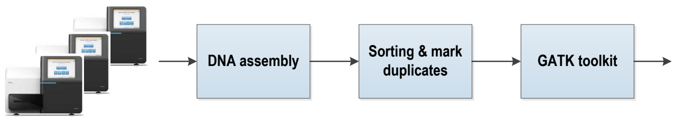

Figure 1. DNA analysis and variant calling pipeline.

genome and reconstruct the sequenced genome from the short read sequences, a process that is called *DNA assembly*. A well-known program used to perform this step is the Burrows Wheel Aligner, typically using the BWA-MEM algorithm. The results of this assembly step are stored in a file with the SAM file format.

After assembly, the short reads in the SAM file are sorted and duplicated reads are marked to avoid using them in downstream analysis. This reduces the bias effect resulting from the way the DNA sample is prepared and sequenced. A widely-used tool to perform this step is the Picard tool. The output of this tool is a compressed file format called BAM. Subsequently, a number of steps are performed to refine the read data and finally to identify the DNA mutations (or so-called variant calling). GATK is a commonly-used tool set for this type of analysis that contains the following tools: Indel Realignment, Base Recalibration and the Haplotype Caller. The output of this tool set is a VCF (variant call format) file with the variations in the sequenced DNA. These widely-used tools described here make part of the Broad best practices pipeline, widely used for variant calling both in research and in the clinic [Auwera13]. We will use the Best Practices pipeline for demonstrating the capabilities of our Spark-based framework, though the framework is able to parallelize tools used in other variant calling pipelines as well.

While the BWA-MEM DNA alignment tool of the best practices pipeline can scale well on a multicore system using multi-threading, it is difficult to exploit parallelism for other tools in the rest of the pipeline. In addition, other pipelines, not as well constructed and optimized as the best practices pipeline, are much less able to utilize the available computational infrastructure efficiently. The objective of our framework is to provide a generic method that is easy to use to ensure scalability to many of the various genomics analysis pipelines out there.

Our framework tackles this challenge by using the Apache Spark big data infrastructure, as shown in Figure 2. In a typical DNA sequencing experiment, two FASTQ files are generated, that represent the 2 ends of a pair of sequences. These 2 input FASTQ files are divided into interleaved chunks using the chunks segmentation tool, written in Scala and using Spark, that also uploads these files into HDFS, thereby making them accessible to all nodes in a distributed cluster. Each chunk is then processed by a separate BWA-MEM task that performs DNA alignment of the short reads against a reference genome. The output of this step represents a list of read alignments (so-called SAM records) that would normally be stored in the SAM file. These records are then read into <key, value> pairs in the memory and regrouped into sub-chromosomal regions by a load balancer according to the number of reads per chromosome. This ensures a better distribution of the subsequent tasks and a better utilization of the computational

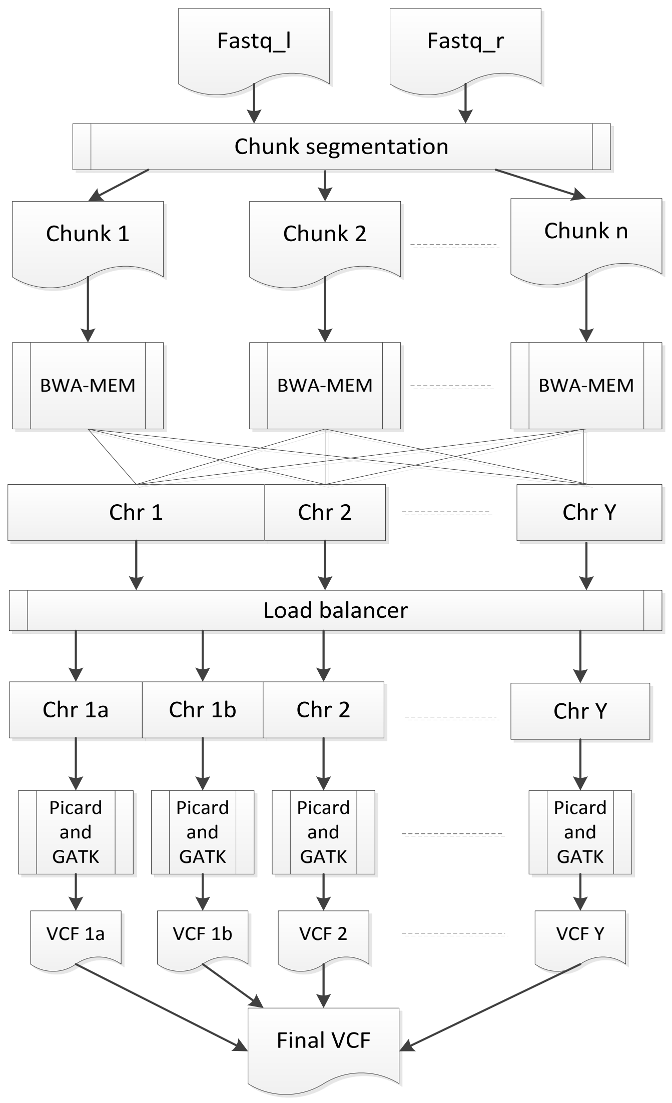

Figure 2. Data flow of the GATK pipeline using our framework

cluster.

Subsequently, these records are sorted using the position of the aligned reads on each chromosome region (which replaces the Picard sort step in the pipeline). Then the rest of the GATK tool set is performed in parallel by executing it on each chromosomal region separately, resulting in various VCF files. Finally, the content of those VCF files are merged into one with all variants identified in the analysis.

Our tool uses a technique similar to Halvade, but instead of using Hadoop MapReduce, our tool uses Apache Spark. Using Apache Spark has several advantages. Firstly, the code written is much succinct as compared to Hadoop MapReduce. Instead of using a map followed by a reduce step, Spark allows our code to contain a few cascaded map calls subsequently, which are explained in Section III. Secondly, Spark can cache data in the memory of the nodes. This means, we can perform extra steps to improve the performance efficiently. For example, in our case, instead of just depending upon the length of the chromosomes, we divide chromosomal regions by inspecting the number of reads mapped for each chromosome, which results in a more optimized load balancing algorithm.

Table I. COMPARISON OF THE TOOLS USED IN OUR FRAMEWORK

Step	Original GATK best practices	Our framework
Align reads	BWA-MEM	BWA-MEM
SAM to BAM	Picard	Uses Picard library
Sort reads	Picard	Quicksort in Scala
Mark duplicates	Picard	Modified Picard
Indel Realignment	GATK	GATK
Base Recalibration	GATK	GATK
Haplotype Caller	GATK	GATK

```
def balanceLoad (x: (Int, SAMRecord),
    chromosomesInfo: HashMap[Int, (Int, Int)],
   avgn: Int) : Array[(Int, SAMRecord)] =
{
  val limit = avgn * 1.5
  var key = x._1
  val sam = x._2
  val chrInfo = chromosomesInfo(key)
  var output = ArrayBuffer.empty[(Int, SAMRecord)]

  if (chrInfo._1 > limit)
  {
    val beginPos = sam.getAlignmentStart()

    if (beginPos > (chrInfo._2 / 2))
    {
      key = key + NumOfChromosomes
      output.append((key, sam))
    }
    else
    {
      output.append((key, sam))
      val endPos = beginPos + sam.getReadLength()
      if (endPos > (chrInfo._2 / 2))
        output.append((key + NumOfChromosomes, sam))
    }
  }
  else
    output.append((key, sam))

  return output.toArray
}
```

Listing 1. Load balancing approach

III. Implementation

We used Apache Spark to implement our framework to parallelize the GATK best practices pipeline. However, we replace parts of the Picard tool in the pipeline with our own code as shown in Table I. First, instead of using Picard to convert the SAM file produced by BWA-MEM to a BAM file, we use the Picard library within Spark to write the BAM file. This eliminates the need for intermediate SAM file conversions, and allows keeping the SAM records in-memory while being sorted. Similarly, instead of writing a BAM file after mark duplicates, we modified Picard to continue with subsequent Picard steps in-memory before writing the BAM file needed by Indel Realignment.

In the following, we discuss the implementation describing our parallelized GATK approach in Section III-A, followed by our load balancing approach in Section III-B. Next, we discuss our optimizations to perform sorting and variant calling in Section III-C. Lastly, we discuss how we improved Picard preprocessing by allowing in-memory computation in Section III-D.

A. Main algorithm

The main algorithm is shown as Algorithm 1. The input is interleaved paired-end FASTQ files in the form of chunks. These chunks are created using the chunk segmentation utility. These FASTQ chunks are mapped to function bwaRun, as seen from the code on line 5. Each bwaRun task then produces an output that consists of key value pairs, where key is the chromosome number, while value is a SAM record.

```scala
def variantCallWrapper(iter:Iterator[(Int, SAMRecord)]) : Iterator[(Integer, String)] =
{
  var res = Array.empty[(Integer, String)]
  val arrayBuf = ArrayBuffer.empty[SAMRecord]
  var key = 0

  while (iter.hasNext)
  {
    val x = iter.next
    key = x._1
    arrayBuf.append(x._2)
  }

  if (!arrayBuf.isEmpty)
  {
    val samRecordsSorted: Array[SAMRecord] = arrayBuf.toArray
    // Sorting
    implicit val samRecordOrdering = new Ordering[SAMRecord] {
      override def compare(a: SAMRecord, b: SAMRecord) = a.compare(b)
    }
    scala.util.Sorting.quickSort(samRecordsSorted)
    //
    res = variantCall(key, samRecordsSorted)
  }

  res.iterator
}
```

Listing 2. Sorting and variant calling

From this data, we first find the average number of mapped reads (in the form of SAM records) per chromosome (line 8). Next, we calculate the number of mapped reads for each chromosome (line 10). These two parameters are useful for load balancing. On line 12, we create a hash map *chrInfo*, which is keyed by the chromosome number and whose values are tuples, where the first element of the tuple is the number of SAM records while the second element is the chromosome's length. The division of chromosome into regions occurs during the load balancing step (line 15), where we make keys according to chromosome regions rather than just chromosome numbers. There, the chromosomes with larger number of mapped reads are divided into more regions than those with lesser number of mapped reads. This division is done by using the average number of mapped reads per chromosome and the number of mapped reads for a chromosome.

Since, at this moment, we used a cluster with 4 nodes, which can run 28 variant calling tasks in parallel, it was enough to divide a chromosome into maximum of two regions. The way we did this is to see if the number of mapped reads for a chromosome is more than 1.5 of the average number of mapped reads per chromosome. If it is more than that, we divide the chromosome into two regions, otherwise the whole chromosome is considered a region. Therefore, with code at line 15, we get key value pairs, where a key represents a chromosome region, while the value represents a SAM record.

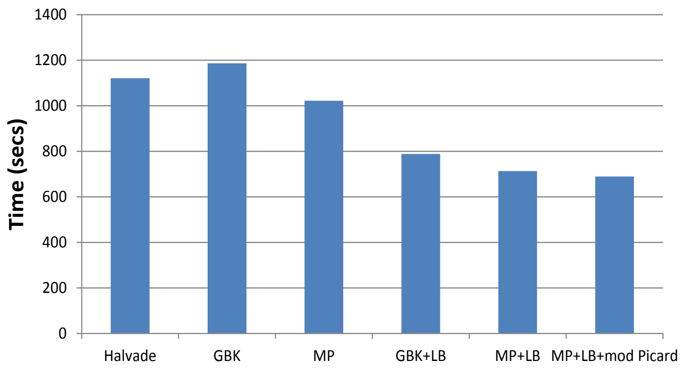

Figure 3. Performance compared to Halvade

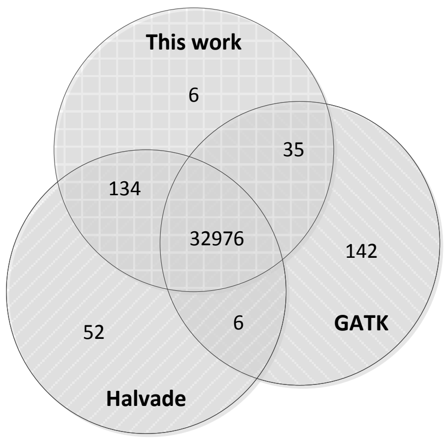

Figure 4. Concordance of variant calls between the selected runs

For doing variant calling, one possibility was to group the SAM records for different keys. This could be achieved by using the *groupByKey* function. We tried it by using the code, *val vcf = chrToSamRecordBal.groupByKey().flatMap(x=> variantCall(x._1, x._2.toArray))*, and performing sorting at the start of the variantCall function. However, we found this approach to be rather slow, as shown in the results in Section IV. Therefore, we used the *mapPartitions* approach, where SAM records are processed in separate partitions. If we had just called mapPartitions, the partitions would have been random, and this means that SAM records from different keys could be in the same partition. To solve this problem, we used a custom partitioner where partitions are made according to the key values. This means SAM records for different keys (chromosome regions) would be in different partitions. The relevant code for doing these steps is shown in line 19 and 20.

Each variant calling task then returns the content of the vcf file produced by the Haplotype caller. Chromosome number is used as the key for each line of the produced vcf file. This helps in sorting the combined vcf file according to the chromosome numbers. The contents of the vcf files produced by all the variant calling tasks are then combined and finally written to a file using the code at line 22.

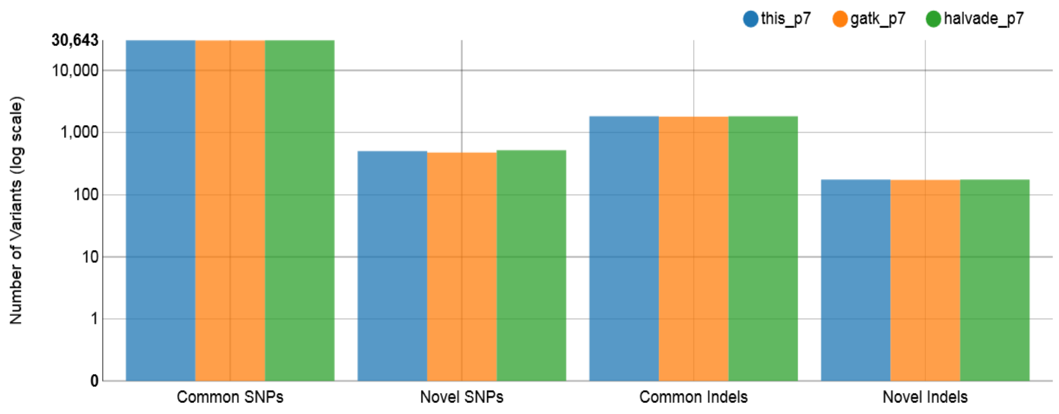

Figure 5. Breakdown of the variant classes by SNPs, Insertions, and Deletions.

B. Load balancing

The load balancing code is shown in Listing 1. A hash map (chromosomeInfo), keyed by the chromosome number, is passed to this function from the main code, where the value of a key contains information for a chromosome in the form of tuples. The first value of that tuple is the number of SAM records for the chromosome, while the second value is the length of the chromosome. If we see that the number of SAM records for a chromosome is greater than the allowed limit, we divide that chromosome into t wo regions, as shown by the code inside the *if* statement on line 10. At line 12, we check the starting position of the SAM record. If the starting position lies at greater than half point of the chromosome, we put it into the right half of the divided chromosome. Otherwise, we check if the SAM record is at the boundary of the divided regions of the chromosome, with its starting position at left side of the divide and the end position at the other side. If that is the case, we copy the SAM record to both the regions, as shown by the code inside the *else* condition on line 19. Lastly, if the end position of the SAM record lies at a position less than the half point of the chromosome, we put that SAM record only in the left half of the divided chromosome. The left half is represented by the original number of the chromosome (key), while the right half is represented by a new number (key + NumOfChromosomes).

C. Sorting and variant calling

The code for sorting and variant calling is shown in Listing 2. Due to this sorting step, there is no need for Picard sorting. Each partition gets a list of tuples, where the first element of the tuple is the key of the region, while the second element is the SAM record. We put those SAM records into an array and then sort them using the quicksort function. These sorted arrays are then used for variant calling. Variant calling for each region then outputs the contents of the VCF file produced by the Haplotype caller. Since chromosome numbers are later used as keys to sort the final combined vcf file, the output array of the variant calling function contains key value pairs of type <chromsome number, line>.

Table II. Speedup compared to Halvade on the 4-node cluster

Configuration	Time [s]	Speedup
Halvade	1121	—
groupByKey	1186	-6%
mapPartitions	1022	10%
groupByKey + load balancing	788	42%
mapPartitions + load balancing	713	57%
mapPartitions + load balancing + modified Picard	689	63%

D. Picard preprocessing

Picard preprocessing, which includes *mark duplicates*, is normally done using different tools, where the output of one tool is given as an input to another tool. This means that unnecessary intermediate files are created, which creates I/O overhead. To circumvent this problem, we combined these tools into one program. The list of SAM records are then just passed around as references. In this way, we avoid unnecessary copying of the SAM records.

IV. Evaluation

We tested the results on two different machines. The first is an IBM Power7+ cluster with 4 nodes. Each node contains 128GB of memory, and two Power7 sockets, each having 8 physical cores and 10MB L3 cache per core. Power7 cores are capable of 4-way simultaneous multi-threading. We compared the results for this machine by running both our Spark implementation as well as Halvade on it. The second machine we used is an identical single-node IBM Power7 system (each core has 4MB L3 cache). We used the illumina-100bp-pe-exome-30x sample data set from Bioplanet [Bio] for all the experiments. In the following, we discuss the performance first followed by the accuracy of the results.

A. Performance analysis

With the sample data set, on the cluster with 4 nodes, Halvade took 1121 seconds to complete, while our implementation only took 689 seconds. Therefore, our approach is 1.63 times faster than Halvade on that machine.

Table II shows the performance of our implementation with different configurations. Without load balancing and using *groupByKey* to group the SAM records and pass them to the

Table III. Speedup compared to the GATK pipeline on the single node system

Configuration	Time [s]	Speedup
GATK pipeline	8725	—
groupByKey	2953	2.95x
groupByKey + load balancing	2779	3.14x
mapPartitions	2224	3.92x
mapPartitions + load balancing	2051	4.25x
mapPartitions + load balancing + modified Picard	1939	4.50x

variant calling step, our framework is slightly slower than Halvade. However, by using *mapPartitions* to group and sort the SAM records for variant calling, our framework is better than Halvade by 10% even without load balancing. When we use the *groupByKey* approach with load balancing, our framework is 42% faster than Halvade, while with *mapPartitions* with load balancing, it is 57% faster. If we also use our modified Picard tool, the performance is improved even further. In that case, it is 63% faster than Halvade.

The information in Table II is also illustrated in Figure 3. In that figure, GBK is an abbreviation for *groupByKey*, while MP is an abbreviation for *mapPartitions*. Moreover, LB is an abbreviation for load balancing.

We also compared our implementation with the multi-threading activated version of the GATK pipeline. For that purpose, we ran our implementation on the single node IBM Power7 machine. Table III shows the performance of our implementation with different configurations. By using *groupByKey* to group the SAM records and pass them to the variant calling step, our framework is 2.95 times faster than the GATK pipeline. By also using load balancing, the speedup increases to 3.14 times faster. However, by using *mapPartitions* to group and sort the SAM records for variant calling, our framework is faster than the GATK pipeline by 3.92 times, even without load balancing. With load balancing, it becomes 4.25 times faster. Moreover, the use of our modified Picard tool improves the performance even further by up to 4.5 times of that of the GATK pipeline. In terms of execution time with the used input data set, the GATK pipeline took 145 minutes, while our optimized implementation took just 32.5 minutes on a single node IBM Power7 machine.

Figure 6 gives a breakup of the execution time for the different tools in the pipeline. The left most bar is for the GATK pipeline, while the middle bar shows the cumulative time of all the tasks of our Spark based implementation for the cluster. Cumulative time is calculated by summing up the execution time of each parallel task. So, basically, its the equivalent of the time that would have been taken by the program if all the tasks were run one by one serially. Lastly, the right most bar shows the breakup using the tasks taking the most time (worst case execution time) for each tool. The sum of these worst case execution times is therefore approximately equal to the total time taken by the program when run on the cluster. We can see here that the summed up execution times of the parallel tasks on the cluster are slightly more than that of the GATK pipeline. However, this is not surprising. For example, for the GATK pipeline, for BWA, we use all the 64 virtual cores of the IBM Power7 machine, while each task of our cluster based approach occupies 8 cores, as the scheduler runs approximately 7 tasks per node. Although we are running 28 tasks on 4 nodes, with each task using the 1/28th data as

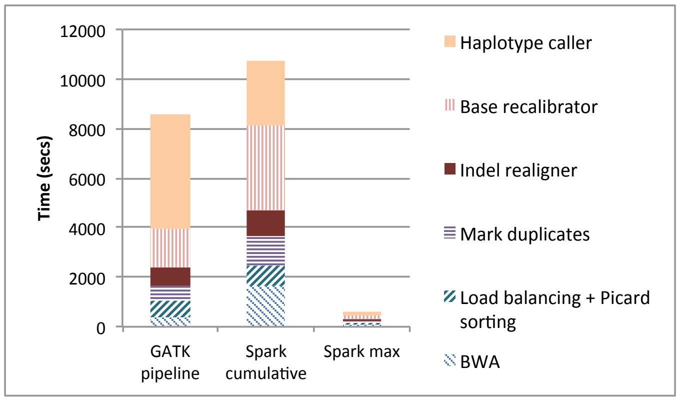

Figure 6. Breakdown of performance.

input compared to the input data of the GATK pipeline, each task is also using fewer resources. Therefore, when we serially add the times of all these tasks, it comes out more. Only if each of those 28 tasks were also using the full 64 cores, the cumulative time for the cluster based approach would have been similar to the time taken by BWA of the GATK pipeline. For the components which do not parallelize well, such as, Mark duplicates and Indel realigner, we do not notice much difference though, which is obvious. In fact, for the Haplotype caller, surprisingly, the cumulative time of the cluster based approach is even less than that taken by the GATK pipeline. The reason of which is yet unknown to us.

B. Accuracy

The accuracy of our solution as compared to the GATK pipeline and Halvade is shown in Figures 4 and 5. The former figure shows the concordance of variant calls between the selected runs while the later shows the breakdown of the variant classes by SNPs, insertions and deletions. From those figures, we can see that our solution achieves an accuracy level comparable with that of the GATK pipeline sharing 99.55% of the variants, as opposed to 99.47% concordance score of Halvade.

V. Conclusions

Next generation sequencing has dramatically increased the speed of DNA sequencing. However, since sequencing produces a large amount of data, post-sequencing DNA analysis requires effective and scalable solutions to ensure high computational performance. For that purpose, we propose a big data framework based on Apache Spark that runs efficiently on a multi-node cluster. Unlike Halvade, which uses the Hadoop MapReduce framework, our tool can make use of in-memory caching provided by Apache Spark, thereby increasing the efficiency and flexibility of the framework and enabling a number of run-time performance optimizations. One such optimization that we implemented is a dynamic load balancing algorithm that divides the chromosomes into regions after the DNA assembly step, depending on the computational load required in each chromosome. Results show that the GATK pipeline with multi-threading activated takes 145 minutes on a single IBM Power7 node, while our approach takes around 32.5 minutes, achieving a speedup of 4.5x. Moreover, on a scalable cluster, our implementation is 63% faster than Halvade on the sample data set used, when executed on a 4-node IBM Power7 big data cluster.

VI. Acknowledgments

Special thanks to Daniel Molnar for implementing the modified Picard program.

References

[Auwera13] G.A. van der Auwera, M. Carneiro, C. Hartl, R. Poplin, G. del Angel, A. Levy-Moonshine, T. Jordan, K. Shakir, D. Roazen, J. Thibault, E. Banks, K. Garimella, D. Altshuler, S. Gabriel, M. DePristo, ”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] Dan Gusfield. 1997. *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge, UK.

[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, J.R. Fitch, Y. Hu, D.J. Corsmeier, H. Zhong, A.N. Wetzel, R.D. Nordquist, D.L. Newsom and P. White,”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.

[Pabinger13] S. Pabinger, A. Dander, M. Fischer, R. Snajder, M. Sperk, M. Efremova, B. Krabichler, M.R. Speicher, J. Zschocke, Z. Trajanoski, ”A survey of tools for variant analysis of next-generation genome sequencing data”, *Brief Bioinformatics*, bbs086v1-bbs086, 2013.