A Study on Optimizing MarkDuplicate in Genome Sequencing Pipeline

Qi Zhao
Computer Science Department
Engineering VI, UCLA
404 Westwood Plaza
Los Angeles, CA 90095
qi.zhao@cs.ucla.edu

ABSTRACT

MarkDuplicate is typically one of the most time-consuming operations in the whole genome sequencing pipeline. Picard tool, which is widely used by biologists to sort reads in genome data and mark duplicate reads in sorted genome data, has relatively low performance on MarkDuplicate due to its single-thread sequential Java implementation, which has caused serious impact on nowadays bioinformatic researches. To accelerate MarkDuplicate in Picard, we present our two-stage optimization solution as a preliminary study on next generation bioinformatic software tools to better serve bioinformatic researches. In the first stage, we improve the original algorithm of tracking optical duplicate reads by eliminating large redundant operations. As a consequence, we achieve up to $50X$ speedup for the second step only and $9.57X$ overall process speedup. At the next stage, we redesign the I/O processing mechanism of MarkDuplicate as transforming between on-disk genome file and in-memory genome data by using ADAM format instead of previous SAM format, and implement cloud-scale MarkDuplicate application by Scala. Our evaluation is performed on top of Spark cluster with 25 worker nodes and Hadoop distributed file system. According to the evaluation results, our cloud-scale MarkDuplicate can provide not only the same output but also better performance compared with the original Picard tool and other existing similar tools. Specifically, among the 13 sets of real whole genome data we used for evaluation at both stages, the best improvement we gain is reducing runtime by 92 hours in total. Average improvement reaches 48.69 decreasing hours.

CCS Concepts

•Applied computing → Bioinformatics; •Computer systems organization → Cloud computing;

ICBRA '18, December 27–29, 2018, Hong Kong, Hong Kong

DOI: https://doi.org/10.1145/3309129.3309134

Keywords

MarkDuplicate; Picard; ADAM; Spark; HDFS

1. INTRODUCTION

DNA sequence [1] represents a single format onto which a broad range of biological phenomena can be projected for high-throughput data collection. DNA sequencing, as shown in Figure 1, is the process of determining the precise order of nucleotides within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases - adenine, guanine, cytosine, and thymine - in a strand of DNA. There are three major steps [2] within the pipeline as the three parts shown from left to right in Figure 1.

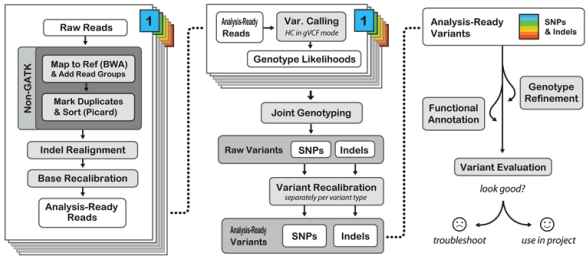


Figure 1: DNA Sequencing Pipeline. [2]

As rapid DNA sequencing methods will greatly accelerate biological and medical research and discovery, the speed of processing the pipeline is important to both biology researchers and medical development. With the help of high-speed processing, we are able to provide more precise medical therapy and in-time clinical diagnosis. However, current DNA sequencing pipeline takes several days to finish only the first step, data pre-processing, on one whole genome data. Therefore, there is a big opportunity to improve the performance of data pre-processing.

There are five major steps [3] in data pre-processing. According to the processing sequence shown in Figure 1, they are BWA-MEM, Sort, MarkDuplicate, IndelRealignment, and BaseRecalibration. BWA-MEM aligns reads to the reference genome by using BWA tool [4]. Sort sorts the alignments by positions on the reference genome by using Samtools. MarkDuplicate distinguishes the duplicate reads and removes them from input by using Picard tool [5]. IndelRealignment performs local realignment to remove the misalignment and BaseRecalibration performs recalibration and

prints reads as well. The last two steps use GATK tool [2]. In order to figure out the which step consumes more time, we break down the runtime as Figure 2 shows.

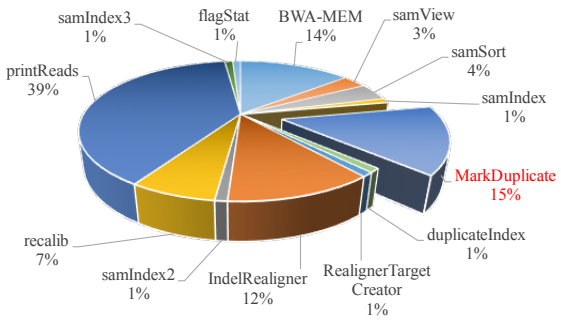


Figure 2: Runtime breakdown of data pre-processing step. [3]

According to Figure 2, the first step, BWA-MEM, consumes 14% time. But it has been well studied by Yu-Ting [3] at 2015. Then we turn to the third step, MarkDuplicate, which consumes the second longest time. The function of MarkDuplicate is to examine aligned reads in the input SAM/BAM file to locate duplicate molecules. All reads are then written to the output file with the duplicate reads flagged or removed. Duplicate reads are non-independent measurements of a sequence sampled from the exact same template of DNA so that they have the same start position. Therefore, we identify duplicate reads based on start position. Duplicate read has to be marked or removed due to it violates assumptions of variant calling and can introduce bias in variant calling. Meanwhile, errors coming from the sample could get propagated to all the duplicate reads as shown in Figure 3. Thus, it will be risky to obtain over-representation in our sequence of areas preferentially amplified. MarkDuplicate does not have a detrimental effect on our overall depth of coverage but increases the quality and reliability of the areas we have covered [2].

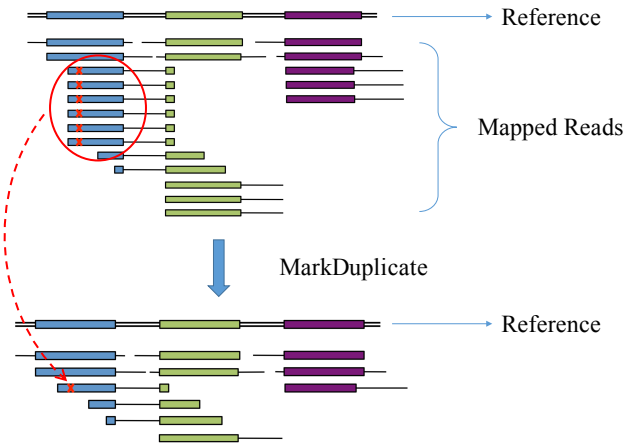


Figure 3: Duplicate reads and MarkDuplicate. [2]

Current application for MarkDuplicate is called Picard, which is developed by Broad Institute for years. Our work first inspects the original MarkDuplicate in Picard and improves one algorithm to reduce the execution time. Then we further profile MarkDuplicate with revised algorithm and narrow the performance bottleneck down to deflate, inflate and I/O operations. Hence, we design a parallel MarkDuplicate at the cloud scale, which eliminates the bottleneck with the ADAM format. Our parallel MarkDuplicate can not only produce the same output as Picard tool, but also be integrated into CS-BWAMEM [3] in the future to scale out the whole data pre-processing.

Our main contributions are the following: 1) We optimize the original MarkDuplicate in Picard, improving the performance of processing whole genome data with the maximal speedup of $9.57X$; 2) We design and implement the cloud scale parallel MarkDuplicate with ADAM format. We shrink the time of I/O processing in MarkDuplicate from 9 hours to 3 hours; 3) We show how ADAM format can be used to scale out and accelerate operations in data pre-processing. With CS-BWAMEM, the whole data pre-processing could be unified to parallel processing.

This paper is organized as follows: Section 2 discusses the algorithm with the problem in detail and our solution as well. Section 3 describes the design of new MarkDuplicate application with supporting parallel I/O processing by applying ADAM format. Each section evaluates the implementation respectively. Section 4 provides the related work of whole genome sequencing and Section 5 presents the conclusions of this paper.

2. PICARD OPTIMIZATION

2.1 MarkDuplicate in Picard

MarkDuplicate in Picard tool is a Java application and it only uses one single thread to process data. It takes sorted BAM file as input and generates new BAM file with duplicate reads marked as output. In addition, the user can customize the configuration to remove duplicate reads in the final output as well. MarkDuplicate is a memory intensive program, which is required in order to detect interchromosomal duplication. To pursue optimization for MarkDuplicate, we should identify the program workflow and the algorithm within it to further understand the implementation of current MarkDuplicate. The high-level processing procedure is identifying duplicate sets first and then finding out the representative read within each duplicate set based on the quality score of each read. A duplicate set is a set of reads in which only one of them is non-duplicate and the rest of them are all duplicate. Quality score is the sum of base qualities for each read, which is already stored in each read as an attribute. The score is pre-calculated before MarkDuplicate. After that, marking all the rest of reads in each duplicate set as duplicate reads by assigning one specific flag in each read. In Picard tool implementation, there are three main steps in total: 1) Load sorted BAM file and iterate each read in order; 2) Go through each input list and build duplicate set; 3) Mark duplicate reads and write non-duplicate reads to output BAM file.

According to Broad Institute, they test MarkDuplicate with 2GB Java heap and 10GB hard memory limit. It takes about 1 hour to process a 8.6GB input data, which contains 63 million reads and 2.5 hours to a 20GB input data, which contains 133 million reads. In order to explore the acceleration opportunity, we test MarkDuplicate in Picard tool with

8GB Java heap and 8GB hard memory limit on a single machine and we use the real whole genome data as well. The result is shown as Table 1.

Table 1: Runtime profiling results.

Data[1]	Size[2]	Time[3]	Time Breakdown		
			Step 1	Step 2	Step 3
798	65G	9.1	3.27	0.10	5.73
799	75G	29.1	3.73	19.1	6.27
800	77G	43.2	3.67	33.2	6.33
801	78G	28.2	3.57	18.2	6.43
802	69G	34.2	3.35	25.12	5.73
803	74G	88.1	3.59	78.43	6.08
804	77G	97.4	3.36	88.4	5.64
805	78G	103.3	3.45	94.13	5.72
806	76G	51.0	3.39	42.67	4.94
807	77G	97.9	3.33	89.4	5.17
808	76G	80.8	3.41	71.1	6.29
809	68G	8.0	3.18	0.25	4.57
810	73G	90.7	3.22	82.7	4.78

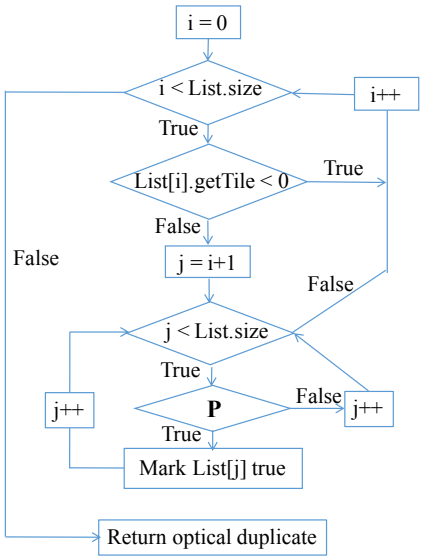
[1] The data number 798 is short for *WGC033798D* as example. The 13 30X coverage paired-end whole genomes are collected from an autism study. These data are 150bp paired-end reads. This set of data is used by us for all the profiling and evaluation through the paper.
[2] The file storage format of this set of data is BAM format, which is the compressed format of SAM format.
[3] The unit of the time in this table is hour.

From the table, we can see that the average time for MarkDuplicate is more than 50 hours except for data 798 and 809. The longest execution time is 103 hours and 20 minutes for data 805. To further profile MarkDuplicate application, we break down the execution to the three steps that we discussed before. The detail is shown as the right part of Table 1. Therefore, we realize that the second step might have the opportunity to improve since it clearly costs more than 50% of the total execution time.

2.2 Problem Analysis

In biology, optical duplicates are sequences from one cluster in fact but identified by software from multiple adjacent clusters. [6] MarkDuplicate collects the number of the optical duplicates and reports the number to the user in the end. We trace the algorithm code and abstract the workflow in Figure 4. Algorithm 1 is the pseudocode. P in Algorithm 1 is the same as the P in Figure 4. In general, there are two nested *for* loops. The outer loop's count i starts from the first read to the end while the inner loop's count j starts from *(i+1)* to the end. Inside the inner loop, the logic is comparing the abstract difference between the Y value of read i and j. If it is larger than the value of *opticalDuplicatePixelDistance*, read j will be identified as optical duplicate. Otherwise, continue the algorithm. Finally the algorithm will return the number of optical duplicates, which is stored in the list called *opticalDuplicateFlags*.

Based on the algorithm, once one read is identified as optical duplicate, it will never be changed again. Therefore, from the algorithm perspective, there are several redundancies within the algorithm that could be eliminated to improve the performance.



P: |List[i].getY() – List[j].getY()|< opticalDuplicatePixelDistance

Figure 4: Tracking optical duplicate read.

Algorithm 1 Track optical duplicate

```
1: for each i ∈ List.size do
2:   if List[i].getTile >= 0 then
3:     for each j ∈ [i + 1, List.size] do
4:       if P == True then
5:         List[j] ← OpticalDuplicate
6:       end if
7:     end for
8:   end if
9: end for
```

2.2.1 Inner Loop Redundancy

As we mentioned, once one read is identified as optical duplicate, it will never be changed again. Thus one possible redundancy could be described as Figure 5. When the current outer loop is at i and current inner loop is at j, which is *(i+1)* according to the algorithm, all the reads in the red area A do not need to compare if they are all optical duplicates already. Under this circumstances, inner loop should start from the position S instead of *(i+1)*.

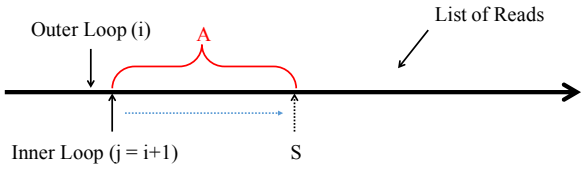


Figure 5: The first redundancy.

2.2.2 Outer Loop Redundancy

The second possible redundancy is shown as Figure 6. P is the criteria for deciding optical duplicate. The meaning of the red and blue arrows are shown in the figure. The term of "false reads" means those reads that are not yet

marked as optical duplicate. Therefore, for one iteration of the outer loop, if all the P in the corresponding inner loop are false, which means all the reads in this inner loop are not optical duplicate, there will be only three situations in total. Noted that the red round brackets indicate the smallest value of the left side of P, which is still larger than *opticalDuplicatePixelDistance*.

As a consequence, we might skip next iteration of the outer loop at *(i+1)* only if the smallest value of the left side of P at *(i+1)* is larger than *opticalDuplicatePixelDistance*. This redundancy should be avoided to gain improvement towards more efficient execution.

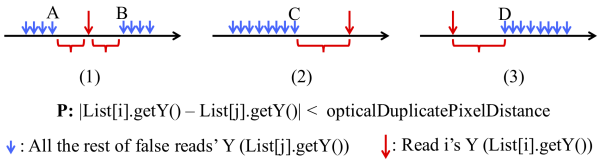


Figure 6: The second redundancy.

2.2.3 *Tail Redundancy*

One more redundancy that we found exists when all the rest of the non-optical duplicate reads are identified as optical duplicate at one iteration of inner loop. Figure 7 shows the detail of this situation. The red area B contains all the reads for inner loop starting from index j. After this iteration, if all the reads in B are optical duplicate, there will be meaningless to continue algorithm.

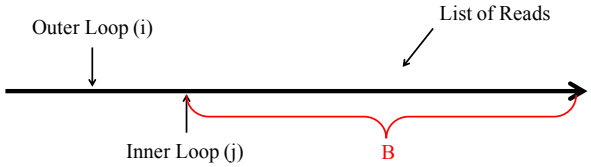


Figure 7: The third redundancy.

We should exit the algorithm and return the optical duplicate list immediately when B are all optical duplicates to remove this redundancy.

2.3 Optimization Design

In general, we create an separate *Arrylist* R to help determining whether there is any of the three kinds of redundancy or not. R stores the index of all the non-optical duplicate reads and R will remove the index when the corresponding read is identified as optical duplicate. For the three redundancy introduced above, we have respective design and implementation. In the end, we update the algorithm as shown in Figure 8 and Algorithm 2.

1. For inner loop redundancy, we will start the inner loop from the larger one between *R[0]* and *(i+1)*. *R[0]* is the smallest index of all non-optical duplicates, which is the position S in Figure 5.

2. For outer loop redundancy, we need to record a check point, which is the Y value of the closest non-optical duplicate to current read i. In Figure 6, it would be the Y value of A and B in *(1)*, C in *(2)* or D in *(3)*. Therefore, we will be able to check whether we need to enter into iteration *i+1* or not. Note that once entering into the inner loop, this check point should be updated while looping.

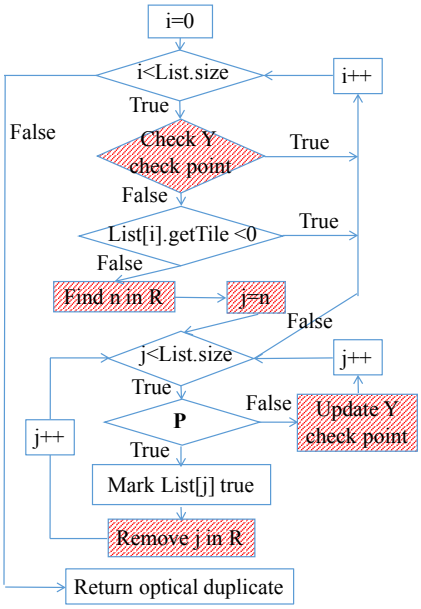


Figure 8: Optimized algorithm work flow.

3. For tail redundancy, we will just simply check whether the R is an empty list or not. If it is empty, return and exit immediately. Otherwise, continue algorithm.

Algorithm 2 Optimized tracking optical duplicate

```
1:  for each i ∈ List.size do
2:    if List[i].getTile >= 0 then
3:      s ← min{i + 1, R[0]}
4:      for each j ∈ [s, List.size] do
5:        if P == True then
6:          List[j] ← OpticalDuplicate
7:          Delete j in R
8:        end if
9:      end for
10:     if R == Null then
11:       return OpticalDuplicate
12:     end if
13:   end if
14: end for
```

2.4 Evaluation

We perform our evaluation for optimized MarkDuplicate on a single machine, which is equipped with Intel Xeon E5-2640 CPU and 128 GB DRAM. We run both the original MarkDuplicate and the optimized version with 8GB Java heap and 8GB hard memory limit. First of all, we take data 807 as an example to perform the evaluation. 807 has more than 1.2 billion reads in total and it costs nearly 3 days to

finish the second step in MarkDuplicate. For our optimized algorithm, it only takes 1.5 hours, which brings us over 50X speedup. Figure 9 shows all the evaluation results of all the 13 sets of whole genome data that we used. All the data listed in the figure have the similar size with *807*.

In Figure 9, we show four time for each data. The description of each time is listed on the bottom. Comparing the first two time we see the improvement on the step 2 only. From the figure, we achieve significant improvement for the second step of MarkDuplicate in Picard tool except for data 798 and 809. The best case we have is reducing the execution time from 94.05 hours to 1.45 hours. The speedup is 64.86X. Even for the worst case, the execution time is still reduced by 17.5 hours and the speedup is 35.31X. The average execution time is reduced to less than 1 hour. The reason for data 798 and 809 is there is no super big list for tracking optical duplicate so that there is no such redundancy to optimize. Since only 2 out of 13 data do not have such redundancy, it is reasonable to assume that this kind of redundancy commonly exists in raw genome data. Comparing the last two time we can see the improvement on the whole MarkDuplicate. Except for data 798 and 809, the minimal time is 29.1 hours while the maximal time is 103.3 hours. After optimization, all the data can be done within around 10 hours. The maximal speedup we gain is 9.65X.

3. CLOUD SCALE MARKDUPLICATE

To explore more optimization, we should consider adopting parallelism to accelerate the processing speed and reduce the execution time as well.

3.1 Motivation

We have successfully reduced the execution time of MarkDuplicate to around 10 hours. Meanwhile, the second step now only costs around 1 hour, which means the rest two steps costs nearly 90% of the total execution time based on the evaluation results shown in Figure 9. The first step is to read all the reads and build two lists for the second step, and the third step is to mark duplicate read and write all reads back to disk file. They are all I/O related operations and become the performance bottleneck in current stage.

The most straightforward solution for parallelizing I/O operation would be partitioning the input data and afterwards processing each partition in parallel. However, it can be implemented unless we eliminate the dependency within the input BAM file data of MarkDuplicate. As we mentioned in the previous section, one of the output lists of the first step, *pairSort*, contains all the paired reads from the input data. There are two kinds of read. One is called fragment read and the other is called paired read. Fragment read is just one single read in the input data. Every read in the input data is fragment read. Nevertheless, some of the reads among fragment read are paired read. They and their pair read form an complete read, which is called paired read. The two pair reads in one paired read are stored in the input data separately. Therefore, while we build *pairSort*, we need to temporarily store all the reads that are paired read if they do not find their pair yet.

As a consequence, we cannot partition the input data directly. Otherwise we will never be able to create the temporary list for the paired read who do not find their pair yet. Without this temporary list, we cannot build *pairSort* and then find out the duplicates. Despite the dependency that we found cannot be removed, we still can design another solution to bypass this dependency and accelerate the I/O processing. Although building two list in the first step cannot be parallelized, reading reads from the input data can be parallelized instead. Mark duplicate read and write reads back to file in the third step can be parallelized as well since there is no dependency found yet. In that way, we are able to accelerate these two steps.

What we need to do is to design a method to handle partitioning the input BAM file data and all the related operations with the partition, including read, write, deflate and inflate. Designing and implementing this framework could be years while we can leverage ADAM [7] to achieve our goal. ADAM is an on-going project leading by AMPLab in University of California, Berkeley. It is a set of formats, APIs, and programming framework for cloud scale genomic processing. It scales efficiently to modern cloud computing performance, which allows us to parallelize genomic data processing. ADAM is implemented on top of Spark [8], Avro [9] and and Parquet [10]. Avro provides explicit data schema access in C/C++/C#, Java/Scala, Python, php and Ruby. Parquet allows access by database systems like Impala and Shark. Spark, a high performance in-memory map-reduce system, improves performance through in-memory caching and reducing disk I/O.

3.2 Algorithm Design

Based on our discussion of acceleration method to reduce the execution time of I/O operation in MarkDuplicate, we design the algorithm shown as Figure 10. Instead of SAM/BAM file format to be the input data file format, now we are using ADAM format as the input data file format. For the single step of MarkDuplicate in the genome sequencing processing pipeline, the execution time should include the data format transforming from SAM/BAM to ADAM when evaluation. However, the trend in this field and our goal are moving towards scaling out data processing to modern cluster and cloud computing in the long term. Thus we will just evaluate our algorithm with ADAM format file as input. Our focus is on whether we can accelerate the process by using another file format.

Our algorithm design is following the processing logic of MarkDuplicate in Picard with using ADAM format file as input and output. The purpose of obeying the processing logic of MarkDuplicate in Picard is to guarantee the validity of the algorithm and the completeness of the output. In Figure 10, operation before "Build two lists" is the first step in Picard. The two operations in the dotted rectangle are the second step in Picard and the rest operations are the third step in Picard. Except for the operations in the dotted rectangle will execute on Driver node of Spark locally, all the other operations will perform parallel processing on cluster.

Following the work flow shown in Figure 10, at first our algorithm abstracts the input ADAM file to an RDD [11], which is resilient distributed dataset in Spark. Then we map this RDD to a new RDD which contains all the reads of the input. The difference is the read in this new RDD only stores the necessary information for marking duplicate. After that, we collect this new RDD back to driver and iterate the reads on driver machine to build that two lists. Also we generate the duplicate indexes on the driver. Once we obtain the duplicate index, we map original input RDD to another new RDD and mark the duplicate reads at the

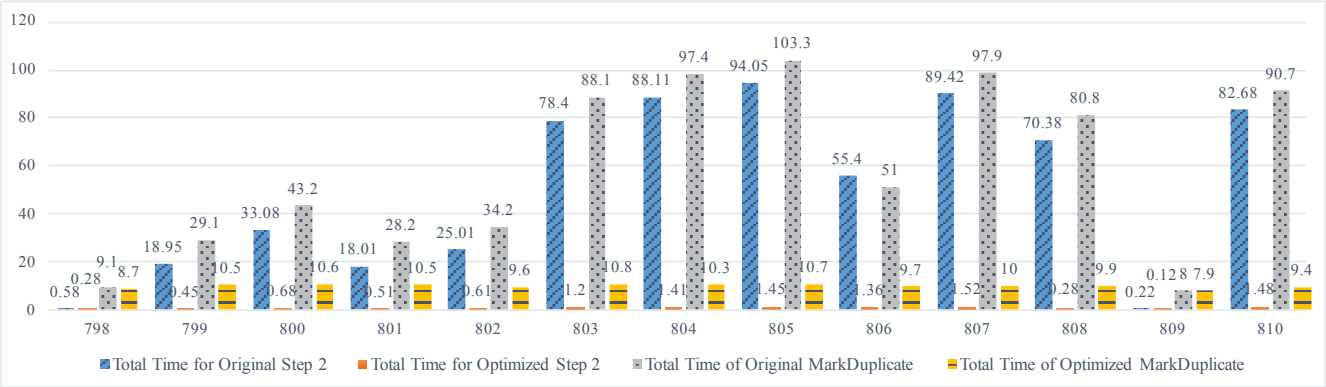


Figure 9: Optimized algorithm performance for whole genome data.

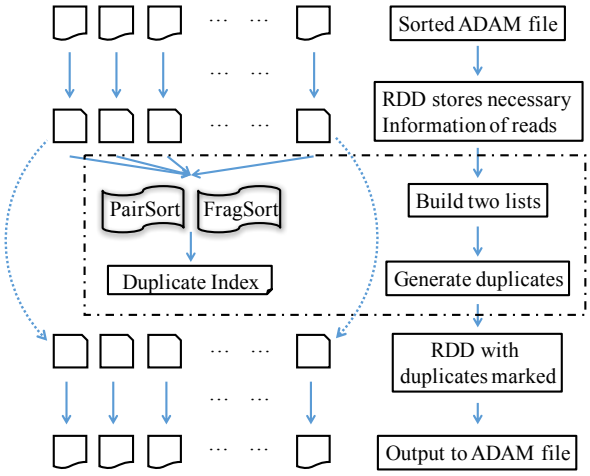


Figure 10: Overview of algorithm design.

same time. At last, we write back the RDD to distributed stored ADAM file. Notice that except for the building *fragSort* that we discussed in section 3.2, all the rest operations are designed as parallel processing and the related data is distributed in its storage. Moreover, All transformations on RDD are lazy, in that they do not compute the results right away. Instead, they just remember the transformations applied to dataset. The transformations are only computed when an action requires a result to be returned to the driver program. This design enables algorithm to be more efficient because we will return only the final result of a chain of transformations, rather than the larger intermedia dataset.

3.3 Implementation

We use Scala [12] to implement our algorithm. For the second step, we directly call the function in Picard while we rewrite the code for the entire first and third step, which deals with distributed data in parallel. During the implementation, we encountered several problems.

3.3.1 Serializable Object

All the objects in Spark have to be serializable. Therefore, we create a new object called *CSAlignmentRecord*. We use this object to store the information of each read for building two lists. However, there is another problem during transforming the read in ADAM to *CSAlignmentRecord*. Some fields do not have the direct corresponding filed in ADAM file. In order to keep the correctness of the transformation, we have to make sure all the data transformed from ADAM format to *CSAlignmentRecord* is exactly matched on biologic meaning. Unfortunately, ADAM does not supply enough detail description on the biologic meaning of those fields. Instead, ADAM do provide a function called *convertADAMtoSam* to support converting one read in ADAM format to SAM/BAM format. Therefore, this problem is solved since all the needed fields can be clearly identified from SAM/BAM format read. The only problem remaining is function *convertADAMtoSam* requires SAM/BAM header as argument. We extract the header from SAM/BAM file and broadcast it to all the worker so that every worker now obtains the header to finish *convertADAMtoSam*.

3.3.2 Memory Limitation

Before iterating reads to build two lists, we have to collect all the reads to driver node in Spark and maintain it with the temporary list containing un-matched pair reads in the memory during the whole iteration. Since the whole genome data used to have more than *1* billion reads, it approximately requires extremely large DRAM, which is unrealistic in practice. One solution can be spill part of the data to disk when memory is insufficient but this will increase huge amount of I/O operation between DRAM and disk. It will harm the performance significantly. Thus, we can leverage the lazy evaluation feature of Spark RDD to collect partial

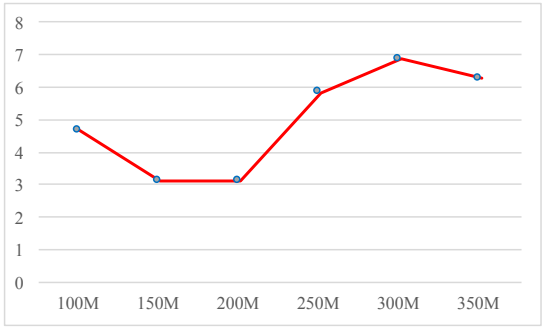


Figure 11: Tuning result for partial RDD size.

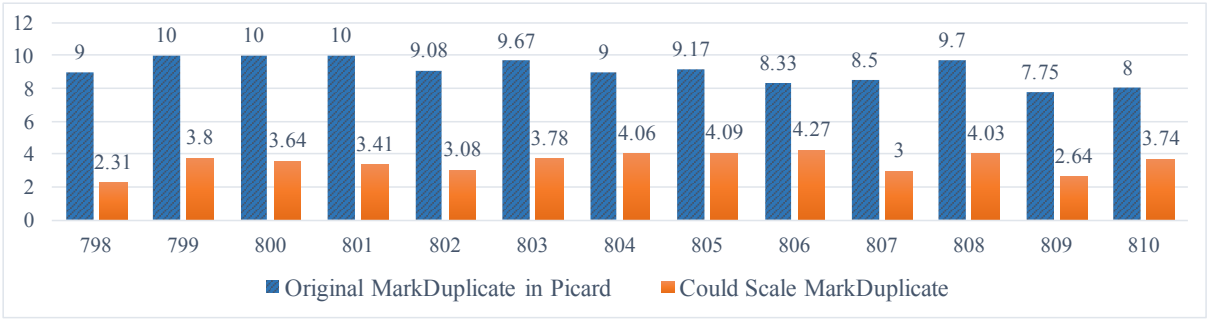


Figure 12: Cloud scale MarkDuplicate performance evaluation result.

reads back to driver's local memory to perform iteration.

The idea is that all the reads are distributed stored in workers' local disk. The RDD of all the reads is abstract presented as a complete and consecutive dataset. Each time, we map and filter out partial set of reads to a new RDD. The order of the partial set of reads is maintained by Spark and we can control the data based on the index of the reads. At last, we just collect the partial set of reads to the memory of driver machine and iterate the reads in memory. Therefore, we tune the partial RDD size for our implementation. We test from 100 million to 350 million reads with using data 800 and get the result in Figure 11, where the y-axis is time in hours and the x-axis is partial RDD size in million. From the figure we notice that the shortest time shows up at 150 million and 200 million. For the following implementation, we use 150 million as the partial RDD size.

3.4 Evaluation

We evaluate our cloud scale MarkDuplicate on our own in-memory cluster. Our cluster uses Spark to perform computation and HDFS [13] as storage infrastructure. More specifically, we deploy Spark 1.5.1 and Hadoop 2.5.2 for HDFS. In our deployment, we deploy both HDFS master node and Spark master node in the same server that we use for the evaluation in section 2. It has two six-core, hyper-threading, Intel Xeon Sandy Bridge processors running at 2.5GHz with 128 GB DRAM. In addition to the master node, we also need slave nodes to store data and perform computation. Our policy is to deploy one HDFS slave node together with one Spark slave node on the same server. It improves the data locality since the Spark slave node can directly fetch data from its local HDFS slave node. For the slave nodes, each server is equipped with two six-core, hyper-threading CPUs and at least 64GB DRAM. Except for master and slave node in Spark, we still need an additional type of node, which is called "driver" node. The Spark driver is where we submit and launch our Spark task. After that, the Spark master will handle resource allocation and task monitoring. Until the resource is sufficient, the launched task will be executed. In our case, the more memory the better performance. Therefore, we select the server with the largest DRAM in our cluster to be our Spark driver. It also has two six-core, hyper-threading Intel Xeon Sandy Bridge CPUs while the DRAM is 256 GB.

In order to compare the performance with the original MarkDuplicate in Picard tool, we keep using the same 13 sets of real whole genome data. For each data, we prepare the sorted ADAM format version in advance and we set 150 million as the size of the partial RDD. The evaluation results are shown in Figure 12.

We extract the performance data from the previous section and put them together in Figure 12 to show the speedup we achieve. The time listed in the table is the total execution time for the first and third step in both Picard and our cloud scale MarkDuplicate. From the figure, we can see that the average time for Picard is around 9 hours. However, even the worst case for our implementation is only 4.27 hours. The best case is just as short as 2.31 hours. In average, it took about 3 hours to finish both reading and writing reads. In the other words, we gain speedup of 3.89X in maximal and 2.7X in average. Theoretically speaking, we reduce the total execution time of MarkDuplicate from roughly 3 days down to less than 3 hours in best case, which saves huge amount of precious time for biologists.

4. RELATED WORK

Picard comprises Java-based command-line utilities that manipulate SAM files for data pre-processing in genome sequencing. Both SAM format and SAM binary (BAM) format are supported. According to [14], Picard MarkDuplicate performs better in removing duplicates than Illumina [15]. There are several other applications performing duplicate marking and removing such as Rmdup in the SAMtools package [16], markdup in Sambamba [17], SAMBLASTER [18] and SEAL [19]. Among them, SEAL is the only distributed tool, which removes duplicates according to the same criteria employed by Picard MarkdDuplicate.

ADAM [7] is a new data storage format and processing pipeline for genomics data. However, their implementation does not have all the output information that Picard has. Thus we implement our own cloud scale MarkDuplicate based on Picard algorithm. Except for ADAM, there are many other platform dealing with next generation sequencing (NGS) [20][21] data. GATK [22] is one of the toolkit that provides MapReduce framework for analyzing NGS data. It contains a small but rich set of data access patterns that encompass the majority of analysis tool needs. The Complete Genomics [23] achieves high accuracy and scalability, which enable complete human genome sequencing in large-scale genetic studies. In [24], they compared the performance of Illumina and Complete Genomics. However, there is still a big development gay between sequencing output and analysis results. Nevertheless, scaling out the NGS is the current trend and faster and consistent platform is needed by bioinformatic researches.

5. CONCLUSION

We propose our two stage optimization to improve the performance of MarkDuplicate in Picard tool. At the first stage, we successfully achieve $5.62X$ speedup on average for the whole MarkDuplicate processing based on our evaluation. To eliminate the performance bottleneck of our improved MarkDuplicate, then we provide cloud-scale MarkDuplicate with the ADAM format disk file instead of the SAM format. Compared to the sequential I/O processing of MarkDuplicate in Picard, our MarkDuplicate finishes within around 3 hours instead of 10 hours. We also show the benefit of using the ADAM format data instead of the SAM format. ADAM supports distributed storage and parallel processing on genome data. Therefore, we obtain the performance improvement by using ADAM as input and output data format. Moreover, using the ADAM format instead of the SAM format gives us the potential opportunity of integrating our cloud-scale MarkDuplicate with other steps in genome sequencing to establish the whole cloud-scale data pre-processing in genome sequencing pipeline, which is the next generation bioinformatic software tool.

6. REFERENCES

[1] J. Shendure and H. Ji, "Next-generation dna sequencing," *Nature biotechnology*, vol. 26, no. 10, pp. 1135–1145, 2008.

[2] "GATK," https://www.broadinstitute.org/gatk/about/.

[3] Y.-T. Chen, J. Cong, S. Li, M. Peto, P. Spellman, P. Wei, and P. Zhou, "Cs-bwamem: A fast and scalable read aligner at the cloud scale for whole genome sequencing," *High Throughput Sequencing Algorithms and Applications (HITSEQ) Poster Session*, 2015.

[4] H. Li, "Aligning sequence reads, clone sequences and assembly contigs with bwa-mem," *arXiv preprint arXiv:1303.3997*, 2013.

[5] "Picard Toolkit," http://broadinstitute.github.io/picard/.

[6] N. Whiteford, T. Skelly, C. Curtis, M. E. Ritchie, A. Löhr, A. W. Zaranek, I. Abnizova, and C. Brown, "Swift: primary data analysis for the illumina solexa sequencing platform," *Bioinformatics*, vol. 25, no. 17, pp. 2194–2199, 2009.

[7] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson, "Adam: Genomics formats and processing patterns for cloud scale computing," *University of California, Berkeley Technical Report, No. UCB/EECS-2013*, vol. 207, 2013.

[8] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–10, 2010.

[9] "Apache Avro," http://avro.apache.org/docs/1.3.0/.

[10] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[11] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[12] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," Tech. Rep., 2004.

[13] D. Borthakur, "Hdfs architecture guide," *https://hadoop.apache.org/docs/r1.2.1/hdfs_design.pdf*, 2008.

[14] H. Xu, X. Luo, J. Qian, X. Pang, J. Song, G. Qian, J. Chen, and S. Chen, "Fastuniq: A fast de novo duplicates removal tool for paired short reads," *PLoS ONE*, vol. 7, no. 12, pp. 1–6, 12 2012. [Online]. Available: http://dx.doi.org/10.1371%2Fjournal.pone.0052249

[15] D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, C. L. Barnes, H. R. Bignell *et al.*, "Accurate whole human genome sequencing using reversible terminator chemistry," *nature*, vol. 456, no. 7218, pp. 53–59, 2008.

[16] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin *et al.*, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009.

[17] A. Tarasov, A. J. Vilella, E. Cuppen, I. J. Nijman, and P. Prins, "Sambamba: fast processing of ngs alignment formats," *Bioinformatics*, vol. 31, no. 12, pp. 2032–2034, 2015.

[18] G. G. Faust and I. M. Hall, "Samblaster: fast duplicate marking and structural variant read extraction," *Bioinformatics*, p. btu314, 2014.

[19] L. Pireddu, S. Leo, and G. Zanetti, "Seal: a distributed short read mapping and duplicate removal tool," *Bioinformatics*, vol. 27, no. 15, pp. 2159–2160, 2011.

[20] M. L. Metzker, "Sequencing technologiesâĂŤthe next generation," *Nature reviews genetics*, vol. 11, no. 1, pp. 31–46, 2010.

[21] J. Zhang, R. Chiodini, A. Badr, and G. Zhang, "The impact of next-generation sequencing on genomics," *Journal of genetics and genomics*, vol. 38, no. 3, pp. 95–109, 2011.

[22] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly *et al.*, "The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[23] R. Drmanac, A. B. Sparks, M. J. Callow, A. L. Halpern, N. L. Burns, B. G. Kermani, P. Carnevali, I. Nazarenko, G. B. Nilsen, G. Yeung *et al.*, "Human genome sequencing using unchained base reads on self-assembling dna nanoarrays," *Science*, vol. 327, no. 5961, pp. 78–81, 2010.

[24] H. Y. Lam, M. J. Clark, R. Chen, R. Chen, G. Natsoulis, M. O'Huallachain, F. E. Dewey, L. Habegger, E. A. Ashley, M. B. Gerstein *et al.*, "Performance comparison of whole-genome sequencing platforms," *Nature biotechnology*, vol. 30, no. 1, pp. 78–82, 2012.