Kafka Interfaces for Composable Streaming Genomics Pipelines

Francesco Versaci
Distributed Computing Group
CRS4, Pula, Italy
francesco.versaci@crs4.it

Luca Pireddu
Distributed Computing Group
CRS4, Pula, Italy
luca.pireddu@crs4.it

Gianluigi Zanetti
Distributed Computing Group
CRS4, Pula, Italy
gianluigi.zanetti@crs4.it

***Abstract*—Modern sequencing machines produce order of a terabyte of data per day, which need subsequently to go through a complex processing pipeline. The conventional workflow begins with a few independent, shared-memory tools, which communicate by means of intermediate files. Given its lack of robustness and scalability, this approach is ill-suited to exploiting the full potential of sequencing in the context of healthcare, where large-scale, population-wide applications are the norm.**

In this work we propose the adoption of stream computing to simplify the genomic resequencing pipeline, boosting its performance and improving its fault-tolerance. We decompose the first steps of the genomic processing in two distinct and specialized modules (preprocessing and alignment) and we loosely compose them via communication through Kafka streams, in order to allow for easy composability and integration in the already-existing YARN-based pipelines. The proposed solution is then experimentally validated on real data and shown to scale almost linearly.

***Keywords*-NGS, Streaming, Flink, Kafka, Hadoop, YARN.**

I. Introduction

Next-generation sequencing (NGS) machines have brought huge improvements in DNA sequencing capacity together with drastic cost reductions, thus paving the way to a myriad of new applications that were previously technologically or economically unfeasible [1]. The output of an NGS machine is, basically, the digitalization of the results of a sequence of massively parallel biochemistry experiments. Converting the raw data thus obtained to biologically relevant information requires various computationally intense processing steps. Therefore, one of the main challenges that needs to be confronted by high-throughput sequencing applications, as would be required by large-scale healthcare applications, is to develop scalable computing tools that can keep up with the massive data generation rate. The standard computational workflows start from the raw data produced by the NGS machines and are based on the use of shared-memory tools, which communicate by means of intermediate files. Traditionally [2], the workflow steps are run on a conventional High-Performance Computing (HPC) infrastructure – a set of computing nodes accessed through a batch queuing system and equipped with a parallel shared storage system. While this is, of course, a working solution, it requires a non-trivial amount of ad-hoc manual intervention to efficiently use the available computational resources and obtain the fast turn-around times that are needed for diagnostic applications [3]. The main issues it presents are how to divide the work of a single job among all computing nodes and how to make the system robust to transient or permanent hardware or software failures. Even though these issues can be surmounted [2], the dependency of the HPC infrastructure on a central shared storage system imposes an intrinsic limitation on the scalability of the platform. In fact, since some of the computational steps are I/O limited, access to storage can quickly become a bottleneck with the increase of the number of computational nodes. As an alternative approach, there has been a recent interest [3]–[7] in supporting the NGS data processing pipelines using completely distributed platforms such as Apache Hadoop. In this approach, the raw and intermediate data files are stored on a scalable file system, like HDFS, and the computation is based on processing steps implemented on the distributed platform – e.g., as Hadoop MapReduce, Spark or Flink programs. While this approach solves the I/O scalability issues and intrinsically provides resilience with respect to node failures, it is still based on the sequential application of the computational steps.

In this work, we explore the adoption of stream computing to help simplify and boost the performance of the genomic resequencing pipeline. Specifically, we decompose the first steps of the genomic processing in two distinct and specialized modules: preprocessing and alignment. We loosely compose these modules via Kafka streams, which also opens to straightforward integration with the already existing Hadoop-based pipelines. To the best of the authors' knowledge, this is the first solution that can process the sequencer's raw data using a fully scalable streaming approach and it is the first example of using a distributed publish-subscribe messaging system (Kafka) to decouple between processing steps of a bioinformatics pipeline.

II. Background

A. The NGS process

The sequencers by Illumina Inc. (http://www.illumina.com) – which are the target of this work – operate by successively attaching a fluorescent identifying molecule to each base of the DNA fragments being sequenced [8]: the genomic material is placed in a *flowcell*, that is organized in

lanes which are, in turn, subdivided in *tiles*. At each cycle, the machine acquires a single base from all the reads by snapping a picture of the flowcell, where a chemiluminescent reaction is taking place. For each picture, and thus cycle, base calling is performed on the image data – mapping each chemiluminescent dot to a specific base (A, C, G, T) based on its color. The process produces base call files (BCL), which contain the bases that were acquired from all the fragments – also known as *reads* – but only in that specific sequencing cycle.

The first step in making sense of the raw data is to reconstruct the original DNA fragments from the "slices" produced by the sequencer, which requires concatenating the elements located at the same positions across several files.

The second step – considering a resequencing experiment – is to *align* or *map* the fragments to the reference genome – i.e., an approximate string matching algorithm is applied to find the position in the reference sequence from which the reads were most likely acquired.

B. Standard practice

In the state-of-the-art, sequencing operations are equipped with a conventional HPC cluster with a shared parallel storage system. Within this context, the standard solution is to perform read reconstruction and demultiplexing using Illumina's own proprietary, open-source tool: bcl2fastq2. It is written in C++, powered by the Boost library [9], and it implements shared-memory parallelism – i.e., it only exploits parallelism within a single computing node. On the other hand, there is variety of alignment programs available and in widespread use [10], [11]. Among these, BWA-MEM [12] is quite popular and has been found to produce some of the best alignments [13], [14]. Like bcl2fastq2 – and the other conventional read alignment programs – BWA-MEM also implements shared-memory parallelism.

The output of BWA-MEM is written as Sequence Alignment Map (SAM) files [15]. SAM is a textual format and rather space inefficient. Thus, SAM files are typically converted to more space-efficient formats, like BAM [15] or CRAM [16] using SAMtools [17]. The CRAM format, adopted in our work, uses reference-based compression to perform efficient lossless data size reduction and it supports, in a flexible and extendable way, a variety of lossy compression algorithms.

C. Apache Flink

The processing of big datasets often involves the application of different algorithmic steps, such as filtering, selecting, pre-processing and so on. A simple way of implementing communication between the different steps while maintaining their independence is by handling the communication via intermediate files: the output files produced by a step become the input for the next one. This approach typically induces a forced serialization of the workflow (a step can begin only when the previous one has completely terminated), as well as difficulties in updating or integrating already processed data. The stream computing paradigm overcomes these problems by establishing a continuous flow of data from the source, to which modules can be attached and detached easily, to build computational workflows that can be updated painlessly.

In recent years numerous stream-oriented big data frameworks have been introduced, such as Spark Streaming [18], Storm [19], Samza [20], Apex [21] and Flink [22] (all of which are Apache projects). We have adopted Apache Flink because of its native streaming capabilities, low latency, exactly-once semantics, powerful computing primitives and clean API.

D. Apache Kafka

With the development of stream computing frameworks, there has been a growing requirement for a means of communication between streaming modules that guarantees low latency, fault-tolerance, scalability and flexibility. Some important projects which try and meet these demands are Amazon Simple Queue Service (SQS) [23], Amazon Kinesis [24] and Apache Kafka [25].

We have chosen to adopt Kafka, which is a message broker based on a publish-subscribe model. In addition to offering the features outlined above, it is also well supported by Apache Flink. Furthermore, starting from version 0.11, it also offers exactly-once semantics, thus guaranteeing that, even in case of failures, each record is eventually delivered to its consumers and that no duplicates are created in the process.

III. Implementation

Our objective is to build scalable, robust and easily composable tools, that enable the processing of continuous streams of sequencing data, while still being able to integrate easily with the already existing distributed analysis tools (of particular interest, the Genome Analysis Toolkit (GATK) [26]). Our pipeline is free software and will soon be released to the public.

A. Data preprocessing

The first module in our pipeline takes care of preprocessing the raw Illumina data, which are generated in the proprietary BCL format. To construct the preprocessor we extended our BCL to FASTQ converter [5] by enabling its output to be sent to a Kafka broker using the built-in Flink-Kafka connector. Schematically, the preprocessor works as follows:

1) Bases and quality scores are decoded and transposed to obtain the reads, which are subsequently sorted by their label (i.e., *demultiplexed*);
2) Tiles are processed in parallel; for each tile, a list of the new Kafka data topics which will be created is first sent to a special control topic; each new data topic corresponds to a logical output;
3) The reads are written into the data topics and an EOS marker is appended to each data topic.

The preprocessor starts sending the reads to Kafka as soon as they are produced – i.e., it does not wait for the entire tile to be processed before starting to send results . Similarly,

TABLE I: Configuration of the Amazon EC2 i3.8xlarge nodes used to evaluate performance.

CPUs	32 virtual cores (Xeon E5-2686 v4, 45 MB cache)
RAM	244 GiB
Disks	4 x 1.9 TB NVMe SSD
Network	10 Gb Ethernet
OS	Ubuntu 16.04.2 LTS (Linux kernel 4.4.0-1013-aws)

the data topics are created incrementally while the input files are processed. In this way modules which are interested in consuming the reads can start processing them as soon as they are produced, without introducing unnecessary latencies – unlike in conventional pipelines which process one dataset at a time.

B. Alignment

The alignment module, implemented from scratch in Flink for this work, exploits our Read Aligner API (RAPI [5]), which in turn relies on a modified version of the standard BWA-MEM aligner [12]. The module consumes reads from the Kafka broker via TCP; thus, the aligner node could be located far from the computation nodes used for the preprocessing step. We conclude the pipeline by writing the aligned reads as CRAM files in an HDFS filesystem. However, the design is modular and extensible, as the data stream can be configured to continue on to additional Kafka brokers and processing modules. This module works as follows:

1) The control topic is polled every 3 seconds, to receive notice of newly opened data topics;
2) The data topics are grouped into Flink jobs which are run in parallel; the reads are passed to the RAPI/BWA-MEM library and aligned;
3) The aligned reads are then written as CRAM files using the Hadoop-BAM library [27].

IV. Evaluation

We evaluated our Flink-Kafka pipeline by comparing its throughput and scalability to those achieved by the conventional pipeline while processing a real human genome dataset. We ran our experiments on the Amazon Elastic Compute Cloud (EC2 – https://aws.amazon.com/ec2) using up to 12 instances of type i3.8xlarge, whose characteristics are summarized in Table I.

The input dataset was produced by an Illumina HiSeq 3000 machine at the CRS4 Sequencing and Genotyping Platform (at the same research organization as the authors of this manuscript). The input dataset contained data from 12 human genomic samples using a single multiplexing tag per fragment; it amounted to 47.8 GB of raw data scattered among 47,050 gzip-compressed BCL files, plus 224 filter files (which contain a QC pass/fail for each read). The size of each uncompressed BCL and filter file is 4.1 MB.

A. Experimental Workflow

To present a fair comparison with the baseline and to have the same setup for all the configurations tested, we chose to have both preprocessor and aligner modules running on the same nodes, at the same time. In more detail, we prepared the following setup, on nodes from Node-1 to Node-n, with $1 \leq n \leq 12$:

- A Kafka broker running on Node-1;
- HDFS distributed among the n computing nodes;
- The preprocessor running on Flink-on-YARN, on the same n nodes;
- The aligner running within Flink in stand-alone mode (i.e., outside of YARN), on all n nodes.

To fully test the streaming paradigm we chose to have preprocessing and alignment running concurrently and thus we tuned the parallelism of the preprocessor in order to maintain, for about half of the total runtime, a feed of new data for the aligner. The data initially reside on the HDFS volume, from which they are read, processed and the output sent via TCP to the Kafka broker running on Node-1; the stream is then read by the aligner jobs.

TABLE II: Running times of our Flink/Kafka pipeline and the baseline on a single node.

Nodes	Time (minutes)
Baseline	137
1	152
2	77
4	39.6
8	20.4
12	14.3

V. Results and discussion

A. Full pipeline

The runtimes obtained by the experiments are shown in Table II. On a single node our pipeline is 11% slower than the baseline, due to the overheads caused by the Flink scheduler and the communication layer. However, as the number of nodes increases, our pipeline achieves near optimal scalability – as shown in Figure 1. The relative scalability (i.e., compared to our runtime on a single node) is 10.6 on 12 nodes, whereas the absolute one (i.e., compared to the faster baseline) is 9.5 on 12 nodes. This result is particularly remarkable since the runtime on 12 nodes is below 15 minutes, and at that level the constant costs of running Flink, YARN and Kafka have a noticeable impact on the total runtime.

B. Limits to scalability

The minimum throughput required to sustain an alignment node at full computation power is about 10 MB/s. We would need 125 nodes to saturate the (full-duplex) network bandwidth of the Kafka broker at this data rate, which provides us with an upper bound for the scalability of the cluster setup used in our experiments.

Since the network becomes our bottleneck when the cluster size increases, if more computation nodes are required to

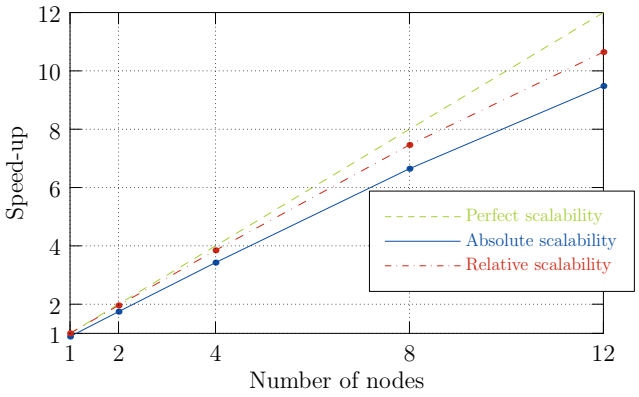

Fig. 1: Strong scaling of our Flink/Kafka pipeline, compared with the single-node baseline.

work in parallel, one could easily add more Kafka brokers to the configuration to have the load automatically distributed among them and thus allow the cluster to grow by (at most) 125 additional alignment nodes per each new Kafka broker.

VI. Conclusion

We have presented a fully scalable streaming resequencing pipeline to process raw Illumina NGS data up to the stage of aligning reads – to the best of the authors' knowledge, the first use of a distributed publish-subscribe messaging system (Kafka) to decouple the processing steps of a bioinformatics pipeline. Although the pipeline described here stops at the aligned reads in CRAM format, it can be easily modified to continue the stream by sending the output to a (possibly different) Kafka broker and then to further processing steps – including variant calling.

Our experiments have shown that the pipeline has very good scalability characteristics, so that an NGS production facility could reasonably expect to reduce their processing time per sequencing run to under an hour using a small commodity cluster. Although scalability has been demonstrated up to 12 nodes, there is circumstantial evidence that, with the same cluster setup and application, a single Kafka broker would be able to support scaling up to about one hundred nodes.

The code presented in this work will soon be available, as free software, at http://github.com/crs4.

Acknowledgments

We thank Gianmauro Cuccuru for the NGS dataset used in the experiments, Massimo Gaggero for his support in setting up the AWS EC2 environment and Brendan Lawlor for interesting discussions on Kafka and its potential uses.

References

[1] J. Shendure and E. L. Aiden, "The expanding scope of DNA sequencing," *Nature biotechnology*, vol. 30, no. 11, pp. 1084–1094, 2012.

[2] O. Spjuth, E. Bongcam-Rudloff, G. C. Hernández, L. Forer, M. Giovacchini, R. V. Guimera, A. Kallio, E. Korpelainen, M. M. Kańduła, M. Krachunov *et al.*, "Experiences with workflows for automating data-intensive bioinformatics," *Biology Direct*, vol. 10, no. 1, pp. 1–12, 2015.

[3] A. Roy, Y. Diao, U. Evani, A. Abhyankar, C. Howarth, R. Le Priol, and T. Bloom, "Massively Parallel Processing of Whole Genome Sequence Data," in *Proceedings of the 2017 ACM International Conference on Management of Data - SIGMOD '17*. New York, New York, USA: ACM Press, 2017, pp. 187–202.

[4] L. Pireddu, S. Leo, and G. Zanetti, "MapReducing a genomic sequencing workflow," in *Proceedings of the second international workshop on MapReduce and its applications*, ser. MapReduce '11. New York, NY, USA: ACM, 2011, pp. 67–74.

[5] F. Versaci, L. Pireddu, and G. Zanetti, "Scalable genomics: From raw data to aligned reads on apache YARN," in *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016*, 2016, pp. 1232–1241.

[6] 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," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-207, Dec 2013. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-207.html

[7] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier, "Halvade: scalable sequence analysis with MapReduce," *Bioinformatics*, vol. 31, no. 15, pp. 2482–2488, aug 2015.

[8] D. R. Bentley *et al.*, "Accurate whole human genome sequencing using reversible terminator chemistry," *Nature*, vol. 456, no. 7218, pp. 53–59, Nov 2008.

[9] B. Schäling, *The boost C++ libraries*. XML Press, 2nd edition, 2013.

[10] H. Li and N. Homer, "A survey of sequence alignment algorithms for next-generation sequencing," *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[11] M. Ruffalo, T. LaFramboise, and M. Koyutürk, "Comparative analysis of algorithms for next-generation sequencing read alignment," *Bioinformatics*, vol. 27, no. 20, pp. 2790–2796, 2011.

[12] H. Li. (2013) Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM.

[13] A. Cornish and C. Guda, "A comparison of variant calling pipelines using genome in a bottle as a reference," *BioMed Research International*, vol. 2015, p. 11, 2015.

[14] S. Hwang, E. Kim, I. Lee, and E. M. Marcotte, "Systematic comparison of variant calling pipelines using gold standard personal exome variants," *Scientific Reports*, vol. 5, p. 17875, Dec 2015, article.

[15] "SAM/BAM format specifications." [Online]. Available: https://samtools.github.io/hts-specs/

[16] "CRAM format specifications." [Online]. Available: http://www.ebi.ac.uk/ena/software/cram-toolkit

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

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York: ACM, 2013, pp. 423–438.

[19] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[20] "Apache Samza." [Online]. Available: http://samza.apache.org/

[21] "Apache Apex." [Online]. Available: http://apex.apache.org/

[22] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[23] "Amazon SQS." [Online]. Available: https://aws.amazon.com/sqs/

[24] "Amazon Kinesis." [Online]. Available: https://aws.amazon.com/kinesis/

[25] "Apache Kafka." [Online]. Available: http://kafka.apache.org/

[26] A. McKenna, M. Hanna, E. Banks *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.

[27] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko, "Hadoop-bam: directly manipulating next generation sequencing data in the cloud," *Bioinformatics*, vol. 28, no. 6, pp. 876–877, 2012.