Flink in Genomics

Efficient and scalable processing of raw Illumina BCL data

F. Versaci L. Pireddu G. Zanetti

– FlinkForward 2016 –

13 September 2016

Outline

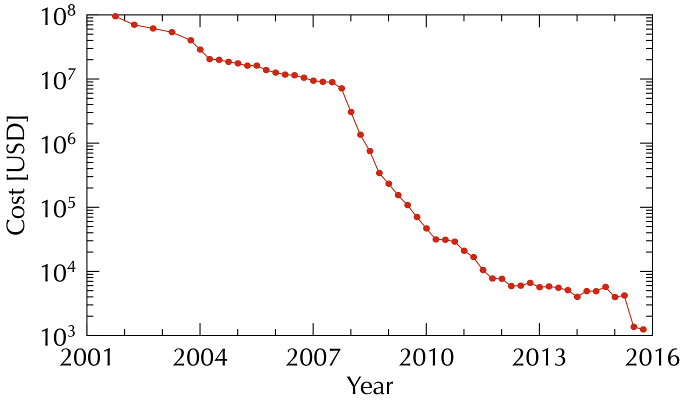
CRS4

Centro di Ricerca, Sviluppo e Studi Superiori in Sardegna

- Research center in Sardinia, Italy
- Focus on big data, biosciences, HPC, visual computing, energy and environment

Next-Generation Sequencing
Cost

- Genome sequencing is now much cheaper than in the past
- About 1000 euros per whole human genome



(Data from `https://www.genome.gov/sequencingcosts/`)

High-throughput DNA sequencing has many applications, including

- Research into understanding human genetic diseases
- Medicine, e.g., oncology, clinical pathology, . . .
- Human phylogeny
- Personalized diagnostic applications

Huge amount of data

A single sequencer can produce 1 TB/day of data

- Which need to be converted, filtered, aggregated, reconstructed, analysed, . . .

Standard pipeline

When using Illumina sequencers, the standard pipeline starts with two programs:

bcl2fastq2 Proprietary, open-source tool by Illumina to convert raw BCL data to FASTQ format

BWA-MEM Free (GPLv3) aligner to reconstruct the full genomic sequence based on the short reads generated by the sequencer

Problem

- Parallel tools, but shared-memory (single node)
- To exploit more nodes data need to be distributed, there can be failures, etc.

BCL converter

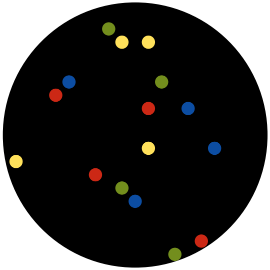
In this talk we present a distributed-memory BCL converter

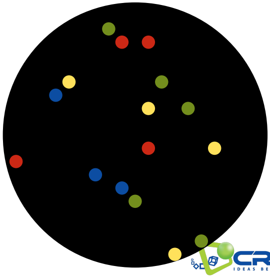
- Developed within the Flink framework
- Written in Scala
- Efficient (i.e., speed comparable to bcl2fastq2)
- Scalable
- Can easily be integrated into existent Hadoop/YARN workflows

Outline

Shotgun genome sequencing

- The DNA is a sequence of four bases: Adenine, Cytosine, Guanine and Thymine (A, C, G and T)
- To reconstruct it, the genome is broken up into short fragments (reads)
- The fragments are attached to a support (tile)
- Fluorescent molecules are iteratively attached to bases of the DNA fragments being sequenced
- At each cycle, the machine acquires (optically) a single base from all the fragments



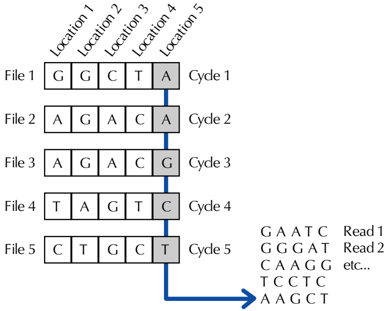


File organization

- We adopt the file structure of Illumina HiSeq 3000/4000 machines
- A single file refers to data obtained by specific lane, tile and cycle combination
- E.g., file `L003/C80.1/s_3_1213.bcl.gz` corresponds to data read from tile $t = 1213$, in lane $l = 3$ during cycle $c = 80$
- Test dataset: 8 lanes $\times$ 112 tiles/lane $\times$ 210 cycles $=$ 188,160 gzip-compressed BCL files $=$ about 250 GB

BCL to FASTQ conversion

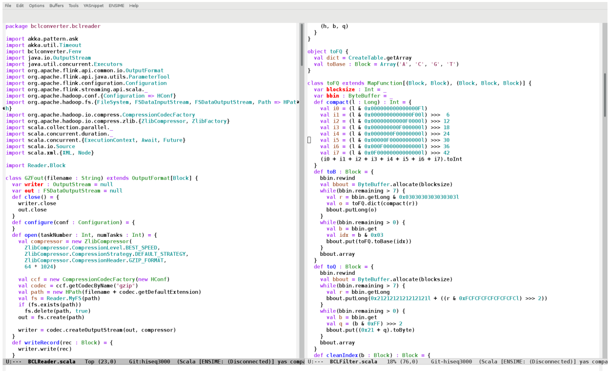
- BCL files are arrays of bytes
- Each byte encodes a base (bits 0-1) and a quality score (bits 2-7)
- `.filter` files specify which reads should be ignored
- `.locs` files contain some metadata which need to be attached to each read
- To get the reads from the raw BCL data we need to perform some sort of matrix transposition



Outline

Implementation choices

- The converter is written in Scala
- We use sbt to handle compilation and dependencies
- All the code is less than 1000 lines
- No fancy IDEs, just EMACS as editor



```
package bclconverter.bclreader

import akka.pattern.ask
import akka.util.Timeout
import bclconverter.Fenv
import java.io.OutputStream
import java.util.concurrent.Executors
import org.apache.flink.api.common.io.OutputFormat
import org.apache.flink.api.java.utils.ParameterTool
import org.apache.flink.configuration.Configuration
import org.apache.flink.streaming.api.scala._
import org.apache.hadoop.conf.{Configuration => HConf}
import org.apache.hadoop.fs.{FileSystem, FSDataInputStream, FSDataOutputStream, Path => HPat
h}
import org.apache.hadoop.io.compress.CompressionCodecFactory
import org.apache.hadoop.io.compress.zlib.{ZlibCompressor, ZlibFactory}
import scala.collection.parallel._
import scala.concurrent.duration._
import scala.concurrent.{ExecutionContext, Await, Future}
import scala.io.Source
import scala.xml.{XML, Node}

import Reader.Block

class GZFout(filename : String) extends OutputFormat[Block] {
  var writer : OutputStream = null
  var out : FSDataOutputStream = null
  def close() = {
    writer.close
    out.close
  }
  def configure(conf : Configuration) = {
  }
  def open(taskNumber : Int, numTasks : Int) = {
    val compressor = new ZlibCompressor(
      ZlibCompressor.CompressionLevel.BEST_SPEED,
      ZlibCompressor.CompressionStrategy.DEFAULT_STRATEGY,
      ZlibCompressor.CompressionHeader.GZIP_FORMAT,
      64 * 1024)

    val ccf = new CompressionCodecFactory(new HConf)
    val codec = ccf.getCodecByName("gzip")
    val path = new HPath(filename + codec.getDefaultExtension)
    val fs = Reader.MyFS(path)
    if (fs.exists(path))
      fs.delete(path, true)
    out = fs.create(path)

    writer = codec.createOutputStream(out, compressor)
  }
  def writeRecord(rec : Block) = {
    writer.write(rec)
  }
U:---  BCLReader.scala   Top (23,0)    Git:hiseq3000  (Scala [ENSIME: (Disconnected)] yas compa
```

```
    (h, b, q)
  }
}

object toFQ {
  val dict = CreateTable.getArray
  val toBase : Block = Array('A', 'C', 'G', 'T')
}

class toFQ extends MapFunction[(Block, Block), (Block, Block, Block)] {
  var blocksize : Int = _
  var bbin : ByteBuffer = _
  def compact(l : Long) : Int = {
    val i0 = (l & 0x000000000000000Fl)
    val i1 = (l & 0x0000000000000F00l) >>>  6
    val i2 = (l & 0x00000000000F0000l) >>> 12
    val i3 = (l & 0x000000000F000000l) >>> 18
    val i4 = (l & 0x0000000F00000000l) >>> 24
    val i5 = (l & 0x00000F0000000000l) >>> 30
    val i6 = (l & 0x000F000000000000l) >>> 36
    val i7 = (l & 0x0F00000000000000l) >>> 42
    (i0 + i1 + i2 + i3 + i4 + i5 + i6 + i7).toInt
  }
  def toB : Block = {
    bbin.rewind
    val bbout = ByteBuffer.allocate(blocksize)
    while(bbin.remaining > 7) {
      val r = bbin.getLong & 0x0303030303030303l
      val o = toFQ.dict(compact(r))
      bbout.putLong(o)
    }
    while(bbin.remaining > 0) {
      val b = bbin.get
      val idx = b & 0x03
      bbout.put(toFQ.toBase(idx))
    }
    bbout.array
  }
  def toQ : Block = {
    bbin.rewind
    val bbout = ByteBuffer.allocate(blocksize)
    while(bbin.remaining > 7) {
      val r = bbin.getLong
      bbout.putLong(0x2121212121212121l + ((r & 0xFCFCFCFCFCFCFCFCl) >>> 2))
    }
    while(bbin.remaining > 0) {
      val b = bbin.get
      val q = (b & 0xFF) >>> 2
      bbout.put((0x21 + q).toByte)
    }
    bbout.array
  }
  def cleanIndex(b : Block) : Block = {
U:---  BCLFilter.scala   18% (76,0)    Git-hiseq3000  (Scala [ENSIME: (Disconnected)] yas compa
```

Algorithmic overview

For each lane/tile combination

- BCL files corresponding to different cycles are opened concurrently
- Bases and quality scores are extracted and filtered
- For each fragment a text header is added, containing various meta-data and an index
- Then they are sorted by their indexes
- Since there can be read errors also in the indices, the repartition is fuzzy: a parameter sets the numbers of allowed misinterpreted symbols
- Finally, a gzip-compressed file for each index is written to disk

DataSet vs DataStream

- Our data is static and read from a storage unit
- This is not a typical streaming application
- We have tried both DataSet and DataStream structures
- Using DataStream is faster
- Because of its better overlap of I/O and computations?

Lesson Learned #1

Try DataStream even if it doesn't seem like a natural fit for your application

Data granularity

- BCL files are arrays of bytes
- It might seem natural to process them in Flink as DataStream[Byte]
- But reading and writing single bytes is not efficient
- We process data in bigger chunks (2048 bytes)
- It imposes a lower load on the streaming framework
- Better cache locality exploitation

Lesson Learned #2

Avoid fine granularity and read in larger chunks

Job granularity

- The job unit (mini-job) is the processing of a lane-tile combination
- Mini-jobs run for about one minute on one core
- We can choose to aggregate n mini-jobs into a Flink job
- And assign c cores to each Flink job
- E.g., aggregate $n = 16$ mini-jobs and run them on $c = 4$ cores
- What about launching one huge Flink job which handles all the work and cores?
- Best results with $n = 2$ and $c = 1$ (with processor $\text{SMT} = 2$)

Lesson Learned #3

Keep Flink jobs reasonably small

Low level optimizations
Use of ByteBuffer

- To extract bases and quality scores we need to perform some bit masks and shifts
- E.g., to get the quality score from byte b we can run

```
val b : Byte = in.get
val q : Byte =
  (0x21 + (b & 0xFC) >>> 2).toByte
```

- We can obtain a 8x speed-up by grouping bytes into 64-bit longs and executing the equivalent operations:

```
val r : Long = in.getLong
val q : Long = 0x2121212121212121l
  + ((r & 0xFCFCFCFCFCFCFCFCl) >>> 2))
```

- To interpret byte arrays as longs, we need to use the ByteBuffer class

Low level optimizations
Use of look-up tables

- We need to convert bases from numeric to ASCII notation
- E.g., `0x0001020303020100` maps to "ACGTTGCA"
- We can do it efficiently by compressing the input and using it as an index in a look-up table
- E.g., `0x0001020303020100` is compressed to index `0b0001101111100100` = `0x1BE4` and searched in the precomputed look-up table
- The table has $2^{16} = 65536$ entries

Scheduling

- To schedule the Flink jobs we use Scala Futures
- Parallel and not blocking

```
// numTasks = number of Flink jobs
implicit val ec = scala.concurrent.ExecutionContext
  .fromExecutor(Executors.newFixedThreadPool(numTasks))
val miniJobs : Seq[MiniJob] = reader.getMiniJobs
// fpar = number of mini-jobs per Flink jobs
val flinkJobs = work.sliding(fpar, fpar)
  .map(fj => Future{runJobs(fj)})
// convert Seq[Future] to Future[Seq]
val flist = Future.sequence(flinkJobs)
scala.concurrent.Await.result(flist, Duration.Inf)
```

Outline

Hardware

- Experiments run on the Amazon Elastic Compute Cloud (EC2)
- Up to 14 instances of r3.8xlarge machines

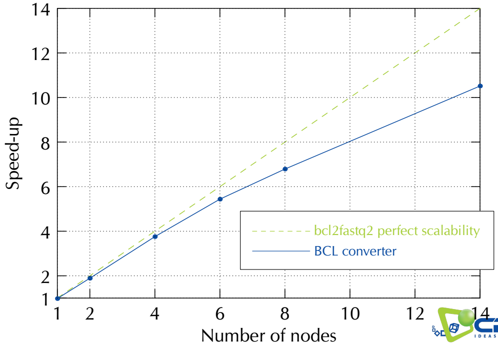
CPUs	32 virtual cores (Xeon E5-2670 v2, 25 MB cache)
RAM	250 GB
Disks	2x320 GB SSD
Network	10 Gb Ethernet

- HDFS distributed among the n computing nodes
- Each datanode using its two SSD disks
- YARN running on the same n nodes
- Flink running inside Hadoop/YARN

Results
Strong, absolute scalability (bcl2fastq2 as baseline)

- Running time of Illumina bcl2fastq2: 57.1 minutes on a single node
- bcl2fastq2 is written in C++ and exploits Boost libraries
- $896 = 64 \cdot 14$ total Flink jobs

Nodes	Time (minutes)
1	58.4
2	30.2
4	15.2
6	10.5
8	8.4
14	5.4



I/O vs Computations

- The program is CPU-bound on the tested hardware
- Total I/O size (input + output): $\approx$ 500 GB
- I/O rate on single node: $\approx$ 150 MB/s
- I/O rate on 14 nodes: $\approx$ 1.6 GB/s
- Note: both input and output are gzip-compressed

Flink features – What we have used

Custom Flink Input/OutputFormat

Hadoop libraries to read/write files

```
import org.apache.hadoop.fs.
  {FileSystem, FSDataInputStream, FSDataOutputStream, Path}
import org.apache.hadoop.io.compress.
  {CompressionCodecFactory, CompressionInputStream}
import org.apache.hadoop.io.compress.zlib.
  {ZlibCompressor, ZlibFactory}
```

DataStream

- `map` and `flatMap`
- `MapFunction` and `FlatMapFunction`
- `filter`
- `split` and `select`

Flink features – What was not available

Efficient zip of DataStreams

Problem

- Given two data streams

```
val names: DataStream[String]
val ages: DataStream[Int]
```

- Join them as

```
val combined: DataStream[(String, Int)]
```

- Useful when reading data about the same object from different files
- E.g., `.bcl`, `.locs` and `.filter` files
- Inverse function of

```
val names = combined.map(_._1)
val ages = combined.map(_._2)
```

Flink features – What was not available

A smarter job scheduler

Remark

We're talking about Flink 1.0: it seems the new job scheduler is much smarter :)

It would be convenient for the job scheduler to be able to

- Pick jobs from some (priority?) queue
- Runs them concurrently on the available Flink task slots
- Start a new job as soon as another one finishes
- Handle failures and retries

Future work

- Integrate our converter into Seal[1] toolkit for short DNA reads manipulation and analisys
- Adopt Flink also in the second stage of the pipeline, i.e., have a Flink-based aligner

[1] `http://biodoop-seal.sourceforge.net/`

Future work

- Integrate our converter into Seal[1] toolkit for short DNA reads manipulation and analisys
- Adopt Flink also in the second stage of the pipeline, i.e., have a Flink-based aligner

Thanks for your attention!

[1] `http://biodoop-seal.sourceforge.net/`