Constructing index structures of biological sequence using Hadoop MapReduce

Bioinformatics-CAP5510, Fall 2012

Rishi Pathak,UFID : 1926-9281,rishi@cise.ufl.edu

INTRODUCTION

DNA sequence alignment is one of the most important application in bioinformatics for identifying sequence similarity,developing homology model of protein structures analysing gene expression, that may be consequence of structural, functional, or evolutionary relationship between the sequences. The genome length varies from few million nucleotides to billion of nucleotides and there are leading algorithms for searching and analysing such large dataset. However these methods rely on precomputed index structures to accelerate the matching and querying without loss of sensitivity and also relieves from exhaustive scanning of every nucleotide. Two of the most important index structures of biological sequence are **suffix array (SA)** and **Burrows Wheeler Transform (BWT).**

GOAL

Advanced algorithm have been developed that constructs suffix array and BWT in linear time, still it requires hours of computation in local environment.
Here I am using **MapReduce** programming model to achieve significant result in distributed environment using its open source implementation **Hadoop.**

BACKGROUND

Suffix Arrays and the BWT

A suffix array is an index that is constructed by lexicographically sorting all the suffixes of the genome sequence. Suffix arrays are easy to construct and significantly improves the sequence alignment task and enable fast lookup. Fig 1.0 shows the suffix array for 8 nt **AGGCTCTA$.** We terminate the sequence with '$'. The first column is the starting position of the associated suffix in second column. If we store the suffixes itself it cost us $O(L^2)$ space requirement, so instead we store the associated suffix offset. Here the SA [8 7 0 5 3 2 1 6 4], stating that the suffix starting at position id lexicographically smallest and so on. Hence total space requirement is 4 L bytes (4 bytes to store each suffix offset)+ L bytes to store the genome itself. So, for human genome of 3 billion nt it takes 12GB for storing suffix offsets and 3GB for the genome, making total 15GB of space requirement.

The closely related BWT reduces this space requirement by a significant amount. Since it is the last column of the BWM, hence it requires space proportional to genome length L. Figure 1.0 shows the BWM and the corresponding BWT of the same reference string.

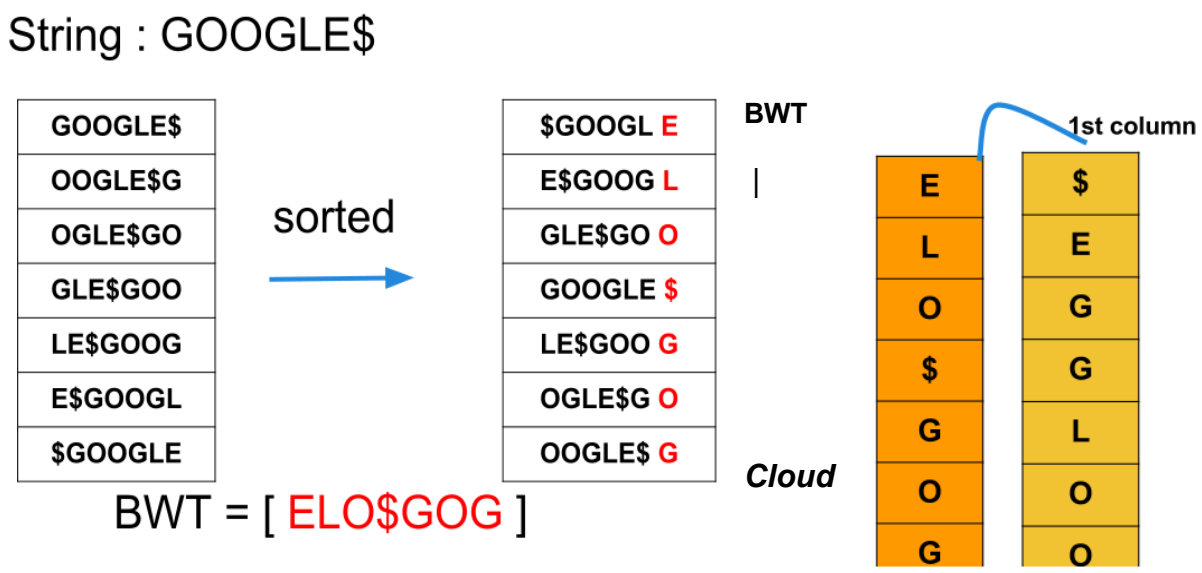
SA	SUFFIX
8	$
7	A$
0	AGGCTCTA$
5	CTA$
3	CTCTA$
2	GCTCTA$
1	GGCTCTA$
6	TA$
4	TCTA$

SA	BWM (sequence)
8	$AGGCTCT A
7	A$AGGCTC T
0	AGGCTCTA $
5	CTA$AGGC T
3	CTCTA$AG G
2	GCTCTA$A G
1	GGCTCTA$ A
6	TA$AGGCT C
4	TCTA$AGG C

Fig 1.0. First table shows the sufix array(SA) and corresponding suffix sorted lexicographically.Second table shows the corresponding BWM consisting lexicographically sorted list of all cyclic rotation of string.BWT is the last column marked red;BWT: AT$TGGACC

BWT can easily constructed once we have the SA, using (BWT[i] = ref[SA[i] − 1]). So, if we have the BWT and the reference string sorted(based on character), then we can easily find a particular sequence based on the knowledge that i^{th} character of BWT is followed by the i^{th} character of the sorted string. So,in Fig 1.1 if we have a string say GOOGLE$, then find its BWT and the if we intend to find GLE then its pretty easy. We go through BWT and find G and check the corresponding character in reference string, which is L and similarly E.

Fig 1.1 : Showing BWT construction



computing and MapReduce

Serial computation of these indexes takes hours of computation. Cloud computing enables to construct them in distributed environment with ease. Cloud computing relieves developer from the overhead of adequate infrastructure. We use MapReduce, parallel programming model which allows computing SA on independent nodes. Hadoop/MapReduce is powerful tool for analysing Big data. MapReduce has 3 phases: Map phase,Shuffle phase,Reduce phase. Map phase scans the whole text and generates intermediate output in key-value form. Shuffle phase sorts and combines values associated with same key and then these keys are partitioned and send to desired reducer which processes the keys and values and generates final output.

INDEX STRUCTURE CONSTRUCTION

The basic approach for constructing suffix array and BWT is to use MapReduce to partition the suffixes into non-overlapping batches and compute the SA and BWT on each of the node for that particular batch/suffix set.

- ***Input :*** The algorithm prepares a file with independent ranges of suffix index/offset for each mapper.So, If you have 'L' length of reference string and 'M' be the number of mappers.Then you have 'M' ranges,such that each mapper processes L/M of the reference string. I distribute the reference string to each of the machine using *DistributedCache.*
- ***Mapper :*** Here I use *NLineInputFormat* class which assign each ranges as a independent map task. In map task you simply emit all the indexes within a particular range from the input file.
- ***Partitioner :*** I am implementing my own partitioner to maintain the order of suffixes and for proper load balancing. I maintain a count of all characters above A,i.e.,count of A and $, similarly count of characters above C,i.e.,count of A+count of C+$, and similarly for G and T and partition the offset in such a way that all suffixes starting with A go to first N reducers based on the count and then all suffixes starting with C go to next O reducers, then all suffixes starting with G go to next P reducers, and all suffixes starting with T go to next Q reducers. So, in all we have M mappers and (N+O+P+Q) reducers. So, for each index, I check the corresponding character in reference string and assign it to desired reducer by decrementing its count and dividing by *perReducer* calculated using length of string divided by number of reducer. So suppose I have sequence *AGGCCTT$,* so I have *Aabove* count as 2(1 A and 1 $),*Cabove*=4, *Gabove*=6 and *Tabove*=8. Now if i encounter a index say 3 so the corresponding character is C and lets assume we plan to run 2 reducers,so *perReducer:8/2=4*. Now we decrement *Cabove* by 1, making

Cabove=3 and divide it by perReducer giving 0,so it goes to *reducer 0*, similarly if we have index 5,corresponding character is T, we decrement *Tablove* by 1 giving *Tabove*=7 and then divide by perReducer.i.e., 7/4=1 (ignoring fractional part), so it goes to *reducer 1.*Similarly for all the indexes we decide the reducer based on the above approach.

- ***Reducer*** **:** In the reducer I store all the indexes in a *TreeMap* which sorts the indexes by comparing their associated suffix from the copy of reference string and I finally emit the suffix array in the *cleanup* function of reducer.

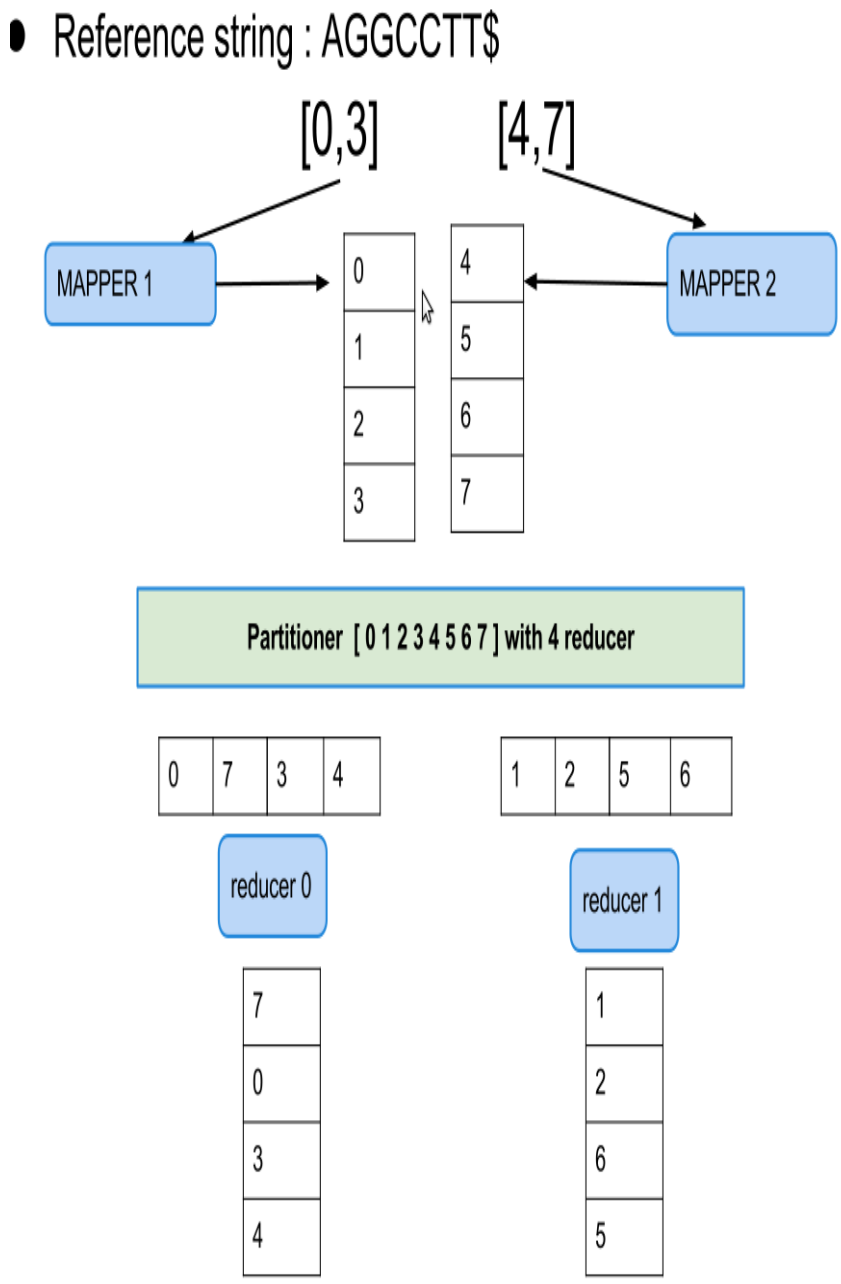
All the above step is same for BWT construction except in reducer where instead emitting SA, I emit the BWT using the formula (BWT[i] = ref[SA[i] − 1]), where *ref* denotes the reference string, and we access its particular character based on SA.

ALGORITHM

```
class Mapper
method map (key , value)
startINdex,endINdex <- split(value,",")
for idx <- startINdex to endINdex do
emit(idx)

class Partition implements partitioner
setConf()
A_count  // no. of A's + 1 ($)
C_count, // no. if A'+C's+1
G_count, //no. if A'+C's+G's+1
T_count              // no. if A'+C's+G's+T's+1
int getPartition(Index,numPartition)
Per_partition = Total/numPartition
switch(n)
case 'A' :           A_count <- A_count - 1
                     A_count/Per_partition
case 'C' : C_count <- C_count - 1
                     C_count/Per_partition
case 'G' :           G_count <- G_count - 1
                     G_count/Per_partition
case 'T' : T_count <- T_count - 1
                     A_count/Per_partition
default :  A_count <- A_count-1
                     return 0;
class Reducer
reducer( index)
TreeMap.put(index)

cleanup()
for all id <- in TreeMap do
emit(id)
```



PERFORMANCE AND RESULTS

Dataset **:** I've taken genome sequences from NCBI(National Center for

Biotechnology Information)

FTP link : ftp.ncbi.nih.gov/genomes/

References : *A View of Cloud Computing, Communication of ACM, 2010. by Michael Armbrust, Armando Fox, Rean Griffith*

Rapid Parallel Genome Indexing with MapReduce,Rohith K. Menon,Goutham P. Bhat,Michael C. Schatz

Using 5 extra large cluster

length of genome(nt)	local mode	distributed mode(AWS) (time in millisecond)
4,686,136 (e coli)	156637	73346
43,261,741 (Oryza sativa)	833789	164480
22,383,619 (Vitis_vinifera)	426909	106655
1,587,646 Bacteria(Bartonella_quintana)	13936	6456
15,072,423 Caenorhabditis_elegans	170179	93875
21,146,708 Drosophila_melanogaster.	416909	117324
35,571,569 Populus_trochocarpa	756439	132542

Using 10 extra large cluster

length of genome(nt)	distributed mode(AWS) (time in millisecond)
4,686,136 (e coli)	51950
43,261,741 (Oryza sativa)	121125
22,383,619 (Vitis_vinifera)	81203
1,587,646 Bacteria(Bartonella_quintana)	4456
15,072,423 Caenorhabditis_elegans	73875
21,146,708 Drosophila_melanogaster.	87324
35,571,569 Populus_trochocarpa	102542