CS466 Class Project: Comparison and implementation of algorithms for Local and Cumulative K-mer statistics on a single machine and a distributed setup (Hadoop)

Abhinav Kohar (aa18) and Sidhartha Satapathy (ss46)

May 4, 2018

Task description

Computation of k-mer statistics is of great importance in bioinformatics and genomics. The frequency of k-mers in a species' genome or in a sequence of genes, acts as a "signature" of that particular genome or sequence of genes. Such a comparison is faster than performing sequence alignment and is also of prime importance in alignment-free sequence analysis. It acts as a preliminary analysis before actual alignment for a variety of tasks like separating species based on mixture of genetic material and estimating genomic size based on k-mer statistics.Wikipedia (2018)

In this work, we would like to compute both local k-mer statistics for a sequence in a set S of sequences, and cumulative k-mer statistics for all sequences in S as a whole. In particular, we want to show how a distributed framework like Hadoop can be leveraged to generate these statistics in a consistent and fast way as compared to sequential algorithms. Although, sequential algorithms are fast enough for commodity hardware, given the amount of genomic data being generated each day, there is an ever increasing need for fast, reliable and distributed algorithms to do k-mer counting.

Furthermore, we implemented the hash-based technique for efficient parallel counting of occurrences of k-mers. We want to see how factors like block size, availability of physical memory on worker nodes, and core available to each worker node on a machine affect the performance of a distributed algorithm on Hadoop.

Background work

Nordberg et al. (2013) presented BioPig, a hadoop based scripting language and analytics

toolkit for rapid prototyping of mapreduce programs, but as so happens the generic nature of the toolkit is also a bottleneck for performance of specific algorithms in our particular case - k-mer counting.

Previously, DSK (Disk streaming of k-mers) Rizk et al. (2013), a streaming algorithm for k-mer counting which uses a fixed amount of memory and disk space has performed well. In addition, KMC algorithm has also been able to do quite well on some datasets, however Jellyfish works the best in terms of memory and runtime as compared to DSK and KMC and hence we decide to compare our distributed approach with the Jellyfish algorithm.

Data and evaluation

We will use Illumina human genome dataset.Rizk et al. (2013) It contains short reads on alphabets A,G,C,T,N Illumina (2010). In order to study scalability of distributed algorithms we need to create artificial datasets of varying lengths from this Illumina dataset. To get datasets for cumulative statistics we just keep on concatenating short reads from Illumina in order they appear till we reach required size. To get a set of sequences for local statistics we can just pick a number between 1 to log (size dataset), for number of sequences. And fill each sequence in this by reads from Illumina in order. This will ensure consistency of results.

We will run both single machine and hadoop version of k-mer statistics and compare the efficiency in terms of speed and memory used. And try to find the breaking point for sequential algorithms.

Hadoop - key factors influencing scalability.

Hadoop is a distributed computing framework, which consists of multiple nodes in a master-slave architecture. Each node has it's own cores, main memory and disk memory (SSD/HDD). The nodes must interact with each other to solve a problem, so key to efficiency is Hadoop the middleware, White (2009).

The design of algorithms in Hadoop, must be done in MapReduce paradigm. This paradigm dictates that the input be split into multiple parts which will be processed by map tasks in parallel. Then, the aggregation of results is done based on reduce tasks.

The hadoop architecture consists of a master node - node manager, which is mainly responsible for global resource allocation and then there are slave nodes which can run multiple worker processes. Both types of nodes have access to HDFS (Hadoop Distributed File System). White (2009)

Given this information, one can see that the number of workers per slave node and the

number and amount of task they can handle are important parameters which will affect the system performance and usually is a trade off between overhead network computation and useful parallel processing. In addition to this, there is a trade off between the number of workers present on a node and the amount of physical memory available, since running time is influenced due to memory thrashing and I/O congestions. One more parameter to consider is HDFS block size. In Hadoop, the input data is partitioned among data nodes which will process it, along with some replication for fault tolerance. This favors data local computation for k-mer statistics. So, block size will determine the number of blocks in a partition of each HDFS files, and is usually the amount of data each map tasks processes. So, block size also affects the number of map tasks in a run which in turn affects the execution time. Small block sizes leads to more time being spent on middle ware management than actual processing of data.

In sum, three parameters must be considered to optimize and fully utilize a Hadoop based algorithmic system :

- Number of workers that can work a slave node at the same time.
- RAM available to each worker
- block size of HDFS

Data statistics

1. The 2GB dataset consists of 8 sequences - adding upto 2 GB created by method described above
2. Similarly, 8GB data consists of 43 sequences.
3. The 32GB data consists of 62 sequences
4. The 64GB data consists of 53 sequences
5. The 128GB data consists of 19 sequences.

Algorithms implemented

Cluster configuration

We implemented distributed algorithms to compute k-mer cumulative statistics (CS) and k-mer local statistics (LS), in Mapreduce paradigm. We setup a Hadoop cluster with 8

worker nodes and 2 node managers each with 8 cores and 32GB RAM each. Each server has 1TB hard-disk space and the hadoop version is HDP 2.7.2. For stability we used azure cloud machines to set Apache Hadoop, and Ambari for monitoring. We do not attach other statistics about our cluster for brevity, but they can be inquired offline (if needed). Other statistics about the cluster and their tuning are explained later in this section.

Input format

The input data described in the above section is read in the form of FASTA files using HADOOP-BAM Niemenmaa et al. (2012).

The Mapper, reducer and shuffling

The mapper scans the input sequence on it's node and maps it to one of the 'r' hash tables it maintains. This first level of summing up by hash tables represents a partitioning of $\sum^k$. As the mapper keeps on extracting k-mers from it's input sequence, the corresponding hash tables get update accordingly. When the map phase ends we send all the hash-tables belonging to the same partition of $\sum^k$ to the same reducer, each reducer is thus independent of others, which aggregates results for that partition. One problem, which we encountered was the out of memory error and array index out of bound exception when the hash table memory block becomes full. So, we alleviated this by flushing the memory block when it becomes full if the mapper was still scanning the input sequence.

During, the shuffle phase a reducer receives all the input for a partition of $\sum^k$ from all hash tables to compute aggregated statistics. They need to be processed one at a time to lower memory requirements. Also, we found out that the memory allocated for each worker at the reducer should be at least twice the memory given to a mapper worker. A target hash table is used at the reducer to sum up all incoming hash-tables at the reducer.

Initialization and setup

In addition, at the beginning we use initialization function in the map phase to create r hash tables with n entries. Both of the parameters are configurable and can be specified by the user at run time. Each mapper, receives a sequence_id and a partial or complete sequence, seq, (partial in case of a long sequence). Also, as we found in our literature su, we converted each character from the input alphabet $\sum = \{A,T,C,G\}$, to a binary encoding. This leads to lower memory requirements as just 2 bits are now needed instead of 8 for the representation. This had the good side effect of improved performance in the shuffle phase when data is being transferred from mappers to reducers, since lower memory requirements lead to lower

I/O and higher throughput.

Finding k-mers - an inspiration from apache lucene

Bit operations can be used now to extract k-mers and improve algorithm efficiency. At the start, a new k-mer is extracted by looking at the first k characters of the sequence and then converting it to it's binary representation. After which the new k-mers are obtained by processing the last k-mer found by using 'SHIFT' and 'AND' bit operations. This is repeated till the end of the sequence is reached or N (ambiguous character) is reached. Each time a k-mer is found it's corresponding hash table entry is incremented.

To identify the entry of the hash table we just use the binary representation of k-mer mod r as the value that identifies this entry in the hash table. Sometimes the mapper may run out of memory for hash tables and may dynamically need more space, in that case we flush out the hash table to the reducer, as discussed previously. Finally the mapper emits a (key,value) pair, where key is sequence_id + hash table id (partition of $\sum^k$), and value is the binary copy of the hash table.

Choice of hash function

Choosing a hash function is an important consideration in this system, since we rely heavily on hashing during both mapper and combiner phase, so efficiency of the system in terms of memory and execution time depends on efficient hashing. We compared standard hash table implementations available in Java with OpenHashMap implementations to keep track of k-mer counts. Provably (Vigna (2017)), OpenHashMap is the most efficient hashing implementation in Java, both in terms of CPU time and memory. Also, OpenHashmap provides functions which can be used to directly update k-mer frequency in a hash table without retrieving it first (typically you need to fetch the value first and then write it in the hash table).

Local statistics

For computing local statistics using mapreduce, since each mapper processes the input from same sequence, it implies that all the frequencies in the hash tables are related to the same sequence. When the mapper emits the (key,value) pair, the value is the binary value of the hash table while the key is a combination of sequence id and hash table id. So, the combiner will be getting all hash tables of partition of the same sequence. So, if we save the reducer output of k-mer counts to different directories, we will have local statistics for different sequences.

All the above steps are summarized in figure 1.

Parameter tuning

One of the biggest challenges for any mapreduce task is finding the right set of parameters for nodes in the cluster to fully utilize the underlying hardware. Now, we will describe how we tuned the parameters we mentioned above:

a.) Number of workers per slave node: As a rule of thumb the number of workers on each node is set equal to the number of cores available. But still tuning this parameter has shown better performance due to the fact that scaling or descaling to larger or smaller number of cores might decrease or increase the performance based on memory thrashing and I/O congestion. So we try to see how number of workers per node as a function of available cores affect the performance and we choose the best numbers. One should also keep in mind that just increasing the number of workers also reduces the memory available per node, and hence mappers cannot process large input blocks.

b.) Block size HDFS: The block size is the amount of input processed by the mapper in one go. So we have two things to consider. First, if the block size is too small, then there will be a large number of mappers to process the input data, and most of the time will be spent in managing these tasks than actual k-mer counting. Second, if we have a large block size then more number of k-mers will be counted by each mapper but it will also lead to an increase of memory requirement for the mapper thus leading to it's interruption. So the block size should be chosen such that the number of mappers is a multiple of number of workers available on a node, and each worker still has sufficient memory to operate on the input. So the block size is analyzed as a function of number of workers.

c.) Number of reducers: The number of reducers will affect the performance as well depending upong how much data they process and their running time. Obviously, in this case the data a reducer gets is a partition of $\sum^k$, so it depends on k. So, we will tune it for a given value of k.

So, given a value of k, we tuned these parameters by running them on different datasets for tuning. So we get a 3-d grid reporting the run times as a function of these parameters. Table 1 shows results for 128 GB dataset.

From this table and using the methods for choosing parameters descibed above we chose, block size of 128MB, with 8 workers and 75 reducers to obtain optimal performance across datasets. Table 1 also proves the need for effective parameter tuning.

We also tried to implement a memory efficient k-mer counting algorithm utilizing efficient hash encoding and instruction level parallelism. But due to lack of time we were not able

Dataset 128GB												
	s=64MB			s=128MB			s=256MB			s=512MB		
# of reducers	# of workers per node			# of workers per node			# of workers per node			# of workers per node		
	2	4	8	2	4	8	2	4	8	2	4	8
15	8	5	7	8	6	7	8	7	12	25	29	X
30	8	6	5	7	5	5	7	5	X	8	9	X
45	9	6	8	5	5	7	5	9	8	9	X	X
60	9	6	5	8	5	5	7	6	8	8	13	X
75	9	5	5	8	5	4	7	5	10	8	9	X
90	9	6	5	8	5	7	8	5	8	9	12	X
105	9	5	4	8	5	5	8	6	X	10	X	X

Table 1: This table above presents the running times of the hadoop algorithm on 128GB dataset, for various values of k (here reported for k=11), along with the number of reducers and worker nodes. We used HDFS block sizes of 64MB, 128MB, 256MB, and 512MB. The running times is in minutes rounded off for brevity and better presentation. 'X' represents the algorithm failed with out of memory errors or array index out of bounds.

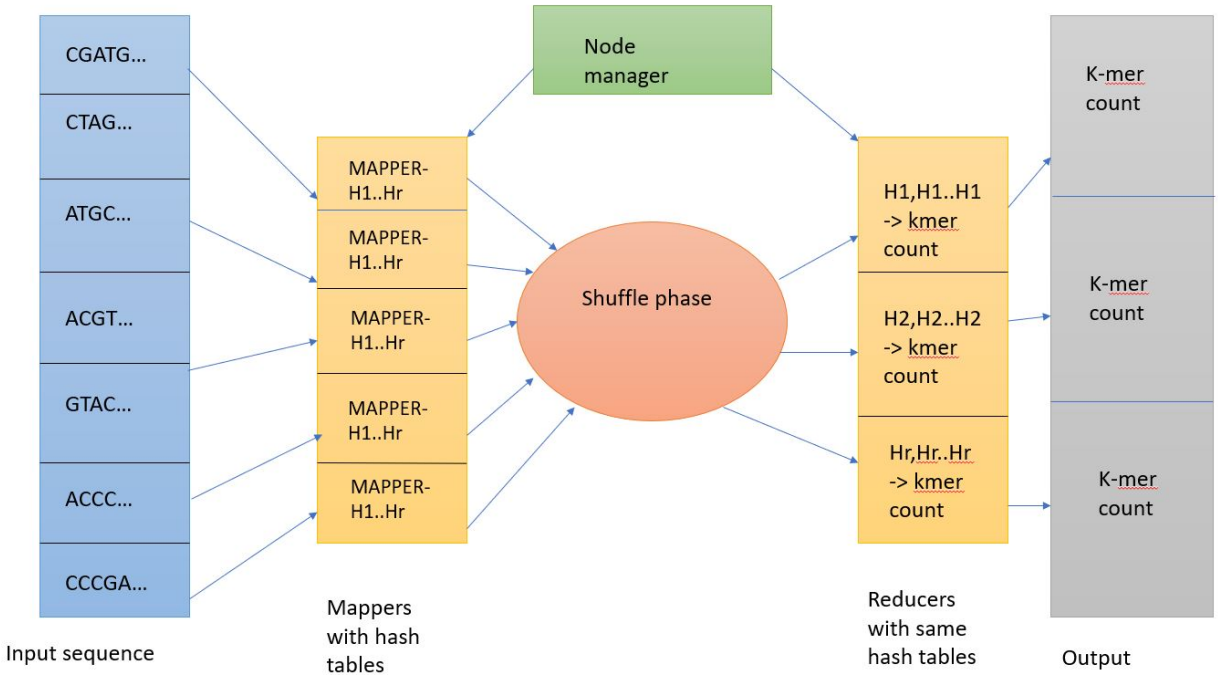
to debug it fully and were getting some incorrect results. So for effective comparison with hadoop algorithm we use the open source version of JellyFish, Marçais and Kingsford (2011). The output k-mers from jelly fish are in binary format for efficiency so we decode them and output as k-mers to compare results with Hadoop algorithm both for correctness and in terms of time. For comparing local statistics with Hadoop version, m subroutine calls are done to JellyFish (for consistency of results so that same underlying implementation is used for both computations) and times are compared.

Results and comparisons

In this section, we will compare the speed of hadoop algorithm with JellyFish, in terms of local and global statistics. We do this for the value of k as 5, and 11, as a representative of both smaller and larger values of k. Similarly we use datasets of sizes 2GB, 8GB, 32 GB, 64 GB and 128 GB. Following figures 3 and 2 summarize our results. (We have used C3js as our charting library c3.)

Jellyfish performs really badly for large datasets and execution time is very high, while hadoop based algorithm scales well. Also, jelly fish works with k less than 31 while our implemetation has no such restrcitions. Also, jelly fish is writetn in C++ while our implementa-

Figure 1: Algorithm



tion is in java. Comparison of underlying java and C++ platforms for these implemenations is beyond the scope of this report.

Why does jellyfish perform badly as compared to Hadoop? Jellyfish is a lightweight, memory efficient, multithreaded hash table for the k-mer counting problem. In Jellyfish, if M is the length of the hash table, the i-th possible location for a given k-mer m is:

$$pos(m,i) = (hash(m) + reprobe(i)) \quad mod \quad M$$

Here quadratic probing is used in case of collisions :$reprobe(i) = i*(i+1)/2$. The lock-free strategy in jellyfish allows for parallel insertions of keys and updates to values (as long as the same value is not updated.) An encoding scheme is also used to reduce the storage required for keys and associated counts.

To allow concurrent update operations on the hash, jellyfish implements a lock-free hash table with open addressing. Such lock-free hash tables exploit the CAS assembly instruction that is present in all modern multi-core CPUs. The CAS instruction updates the value at a memory location provided that the memory location has not been modified by another thread.

For updating a count, Jellyfish finds an appropriate slot using the hash function and then does a CAS operation assuming that the entry in the hash is empty. If the returned value of the CAS operation is either EMPTY or equal to key, then that position is used for storing

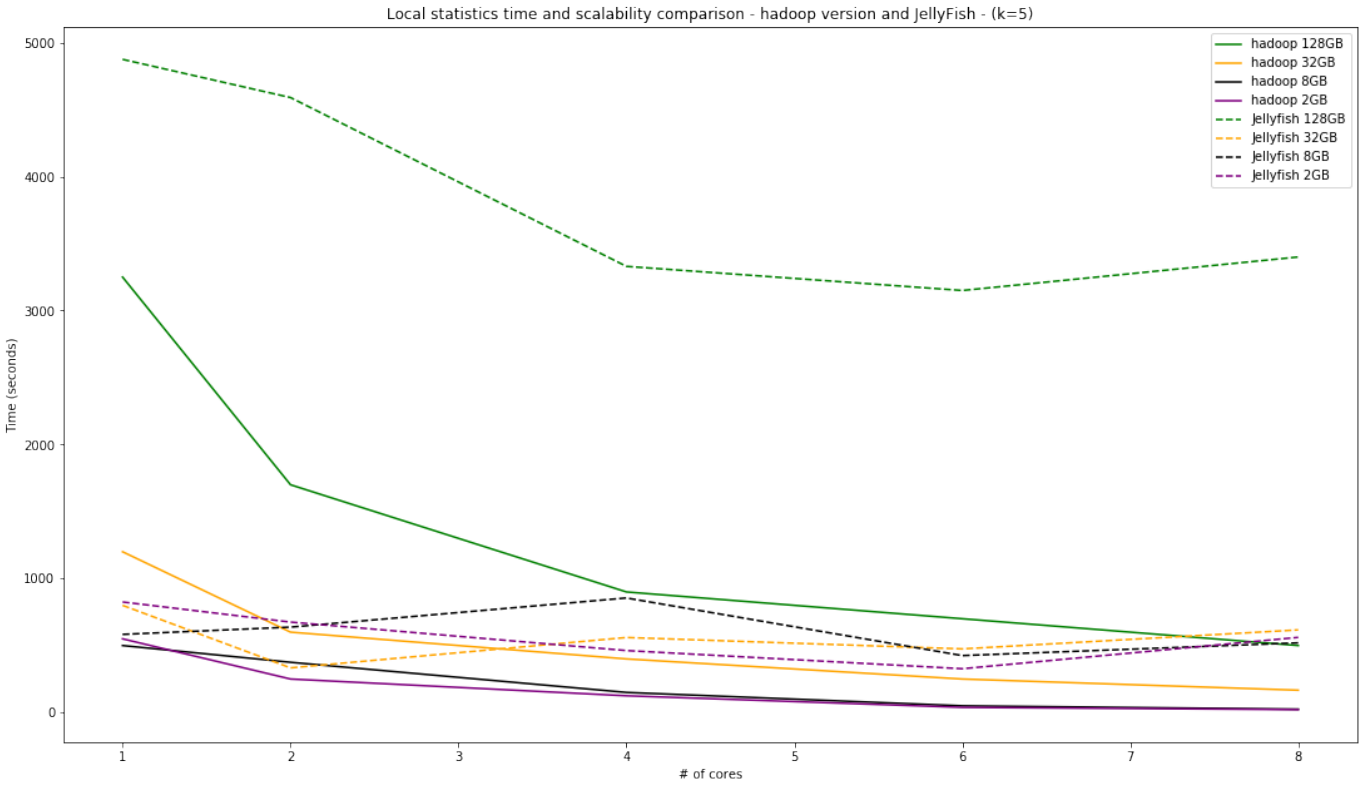


Figure 2: Local statistics results

the key. Otherwise, there is a key collision: the reprobe value is incremented and process start over. The procedure fails if the maximum number of reprobes has been reached.

One important assumption in jellyfish is that, for k-mer counting the required size of the hash table should be easy to estimate or potentially the entire available memory is used. Hence, in the event that the hash table is full, it will be written to disk instead of doubling its size in memory. (a potential bottle neck).

The running time of Jellyfish takes number of operations proportional to $\alpha_k = 2k\lceil 2k/w \rceil$ where w is the length of a machine word, to compute the matrix-vector multiplication, and the time to insert one k-mer in the hash table is proportional to $\alpha_k + max\_reprobe$, Marçais and Kingsford (2011).

In the case where t intermediary hash tables of size s_i, $1 \leq i \leq t$ with $\sum_{i=1}^{t} s_i = n$, were written to the disk, the time to create all t hash tables is $O(n(\alpha_k + max\_reprobe + log(max\_reprobe)))$. So if large amount of memory is available as compared to the data, the number of hash tables created will be constant and run time will be smaller. On the other hand, if amount of memory available is small in comparison to the dataset, the number of hash tables is created is proportional to n and the run time is larger.

So for datasets of size around 32GB and less, the run time of jellyfish for CS is slightly

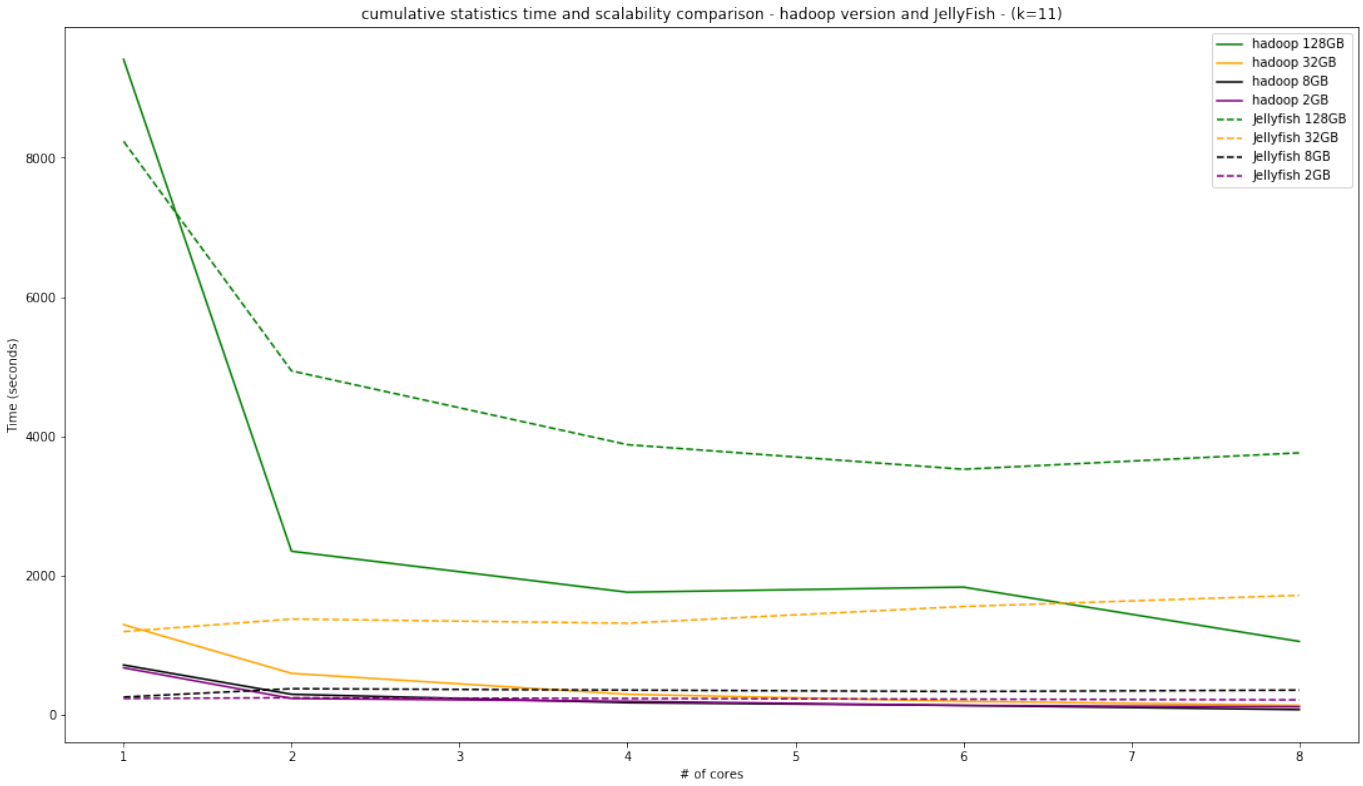


Figure 3: Cumulative statistics results

higher than hadoop version, but for larger datasets (64 GB and 128 GB), due to more collision in hashes and large size hash tables, and merging of those hash tables and intermediate writing to disks, Jellyfish performs really slower than hadoop version both for CS and LS. And as the number of cores increase, the algorithm eventually starts to perform badly compared to hadoop which keeps on performing better, more cores lead to faster processing of input, leading to more collisions and probing and available physical memory becomes a bottleneck and the run time increases because of disk I/O. No such things happens in distributed hadoop algorithm and it can utilize more cores efficiently.

Conclusion

We presented programming choices and architectures for big data in bioinformatics for k-mer counting using Hadoop and show that our implementation is as good as and sometimes(for larger datasets) better than multi threaded state of the art algorithm Jellyfish. We also conclude that design and implementation of algorithms cannot be eluded to Hadoop distributed framework, we still need better parameter tuning and algorithm design(apart from just mapreduce paradigm) to achieve better performance of algorithms.

Contributions and code

For cluster access and results contact us. The cluster is live at:
https://abhinavk.azurehdinsight.net/#/main/services/YARN/summary
Both of us were involved in writing the codes for the hadoop framework as well as the analysis of the results. In addition, Abhinav analyzed the results on jellyfish and Sidhartha wrote a naive hash based algorithm which was taking a very long time to run and is not discussed in the paper. Furthermore, both of us were also involved in writing the paper.

Code is available on gitlab: https://gitlab-beta.engr.illinois.edu/aa18/cs466

References

c3 is a d3-based reusable chart library that enables deeper integration of charts into web applications. URL `https://github.com/c3js/c3`.

Illumina. Illumina human genome dataset. `https://www.illumina.com/informatics/sequencing-data-analysis/data-examples.html`, 2010. [Online; accessed 20-April-2018].

Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics, 27(6):764–770, 2011. doi: 10.1093/bioinformatics/btr011. URL `http://dx.doi.org/10.1093/bioinformatics/btr011`.

Matti Niemenmaa, Aleksi Kallio, André Schumacher, Petri Klemelä, Eija Korpelainen, and Keijo Heljanko. Hadoop-bam: directly manipulating next generation sequencing data in the cloud. Bioinformatics, 28(6):876–877, 2012. doi: 10.1093/bioinformatics/bts054. URL `http://dx.doi.org/10.1093/bioinformatics/bts054`.

Henrik Nordberg, Karan Bhatia, Kai Wang, and Zhong Wang. Biopig: a hadoop-based analytic toolkit for large-scale sequence data. Bioinformatics, 29(23):3014–3019, 2013. doi: 10.1093/bioinformatics/btt528. URL `http://dx.doi.org/10.1093/bioinformatics/btt528`.

Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk. Bioinformatics, 29(5):652–653, 2013. doi: 10.1093/bioinformatics/btt020. URL `http://dx.doi.org/10.1093/bioinformatics/btt020`.

Vigna. Fast util hashmap java. `http://fastutil.di.unimi.it/`, 2017. [Online; accessed 20-April-2018].

Tom White. Hadoop: The Definitive Guide. O'Reilly Media, Inc., 1st edition, 2009. ISBN 0596521979, 9780596521974.

Wikipedia. K-mer — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=K-mer&oldid=835355509, 2018. [Online; accessed 20-April-2018].