Rapid Parallel Genome Indexing with MapReduce

Rohith K. Menon
Department of Computer Science
Stony Brook University
rkmenon@cs.stonybrook.edu

Goutham P. Bhat
Department of Computer Science
Stony Brook University
gbhat@cs.stonybrook.edu

Michael C. Schatz*
Simons Center for Quantitative Biology
Cold Spring Harbor Laboratory
mschatz@cshl.edu

ABSTRACT

Sequence alignment is one of the most important applications in computational biology, and is used for such diverse tasks as identifying homologous proteins, analyzing gene expression, mapping variations between individuals, or assembling *de novo* the genome of organism. Except for trivial cases involving just a small number of short sequences, virtually all other sequence alignment tasks rely on a precomputed index of the sequence to accelerate the alignment. Two of the most important index structures are the suffix array, which consists of the lexicographically sorted list of suffixes of a genome, and the closely related Burrows-Wheeler Transform (BWT), which is a permutation of the genome based on the suffix array. Constructing these structures on large sequences, such as the human genome, requires several hours of serial computation and must be performed for each genome, or genome assembly, to be analyzed. Here we present a novel parallel algorithm for constructing the suffix array and the BWT of a sequence leveraging the unique features of the MapReduce parallel programming model. We demonstrate the performance of the algorithm by greatly accelerating suffix array and BWT construction on five significant genomes using as many as 120 cores leased from the Amazon Elastic Compute Cloud (EC2), reducing the end-to-end runtime from hours to mere minutes. The source code is available under an open source GPL License at:
`http://code.google.com/p/genome-indexing/`

Categories and Subject Descriptors

J.3 [**Life and Medical Sciences**]: Biology and Genetics; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms

*Corresponding Author

MapReduce'11, June 8, 2011, San Jose, California, USA.

Keywords

DNA sequence analysis, parallel suffix array construction, Burrows-Wheeler Transform, MapReduce, cloud computing

1. INTRODUCTION

DNA sequence alignment is an extremely common application in computational biology, in which sequences of the four different nucleotides (nt) are matched or arranged to find regions of maximal similarity [5]. For example, sequence alignment is used find conserved regions between genomes [9, 1], measure transcription [20], and to reconstruct the genome *de novo* [19]. The genome of an organism varies in length from a few million nucleotides for simple bacterial organisms, three billion nucleotides for the human genome, to hundreds of billions of nucleotides for the largest known genomes. Given their length and complexity, the leading methods for searching and analyzing genomes rely on an auxiliary precomputed index structure that allow for rapidly matching and querying without loss of sensitivity and without exhaustively scanning every individual nucleotide.

Two of the most important index structures of biological sequences are the suffix array (SA) [14] and the Burrows-Wheeler Transform (BWT) [2]. A suffix is any substring that extends to the last character in the string, including the degenerate suffix starting at first character of the string and extending to the last. A prefix is any substring that includes the first character of the string. The suffix array consists of the lexicographically sorted list of all suffixes of the string. Because the suffixes are sorted, this simple index allows for rapid binary search algorithms for matching query sequences of any length, and more advanced algorithm even allow differences between the reference and query sequences during the suffix array matching. The closely related BWT reduces the space requirements of the suffix array, which requires $>$ 12 GB for the human genome, by recording as an index a (reversible) permutation of the string based on the ordering of the suffix array. As a permutation, the size of the index is the same size as the string itself, only 3 GB for the human genome.

The suffix array can be naively constructed using a string-comparison based sort of the suffixes, which requires $O(G\ lg\ G)$ comparisons for a genome of length G using a QuickSort algorithm, although each comparison may evaluate $O(G)$ characters raising the overall runtime to $O(G^2\ lg\ G)$. Several advanced methods have since been developed methods that are able to construct the suffix array in linear time [17]. Nevertheless, the leading methods for constructing the BWT or suffix array require several hours of computation on a desk-

top workstation, especially if there is insufficient memory to hold the entire suffix array in memory. Given the critical importance of suffix array construction, parallel methods have also been developed to accelerate the computation over potentially many computers. The leading parallel method, called PDC3, has been developed for MPI-based systems by recursively computing the suffix array of a carefully selected 2/3's of the genome, from which the suffix array of the remaining 1/3 can be inferred in linear time [8]. The recursive nature of the algorithm, however, requires substantial low-latency and high-bandwidth interconnects that are not generally available.

Here we present the first MapReduce [3] parallel algorithms for suffix array and BWT construction. The algorithm uses the inherent data processing capabilities of MapReduce to divide the suffix array construction into multiple independent ranges that are then independently solved. By carefully selecting and ordering the partition points, the ranges are well-balanced across processing cores, and the final output will form a total order to the array. Using our algorithm in experiments with as many as 120-cores leveraging the open-source Hadoop[1] implementation of MapReduce, we have reduced the time to index the human genome from several hours to just a few minutes. By making the source code open-source we expect genomics researchers around the world to benefit from this advance. The remainder of the paper is organized as follows: Section 2 provides additional background material on DNA sequencing, suffix arrays and the BWT, and a brief overview of MapReduce relevant to our algorithm. Section 3 describes a basic MapReduce-based algorithm for suffix array construction, which we refine in Section 4 with several novel techniques that greatly enhance its performance. Section 5 describes the evaluation of our implementation on several important genomes over a variety of cluster configurations. We will conclude in Section 6 with a discussion of the advances made possible by our algorithm and future research directions.

2. BACKGROUND

2.1 Suffix Arrays and the BWT

Sequence alignment is a ubiquitous task in computational biology driven by a multitude of applications and the dramatic rise in DNA sequencing capabilities. Current DNA sequencing machines are generating sequences at a tremendous rate, exceeding 25 GB of sequence data per day per machine, in billions of short ($50-500$ nt) DNA sequences called reads [15]. Depending on the application, those reads will then be aligned or mapped to a reference genome, such as to identify variations between people [10] or measure biological activity in different cells [20], or they will be compared to each other to reconstruct the full length genome in a task called *de novo assembly* [19]. All of these sequence alignment tasks, and many others, benefit from using an index: in the case of read mapping, the index is used to quickly determine where in the genome each read originated; in the case of de novo assembly, an index is created on-the-fly from the intermediate assembly to aid the computation. After assembly an index will be computed to support any further analysis of the newly sequenced genome. As such developing efficient algorithms for indexing is essential. In particular, the recently announced Genome 10K project[2] aims to sequence and assemble the genomes of 10,000 vertebrate organisms, each of which will need to be indexed prior to any analysis.

Table 1: Suffix Array (left) and Burrows-Wheeler Matrix (right) of GATTACA$. The SA field stores the starting position of the lexicographically sorted suffixes of the reference sequence. The BWM consists of the lexicographically sorted list of all of the cyclic rotations of the string. The BWT consists of the last column of the BWM: ACTGA$TA.

index	SA	suffix
0	7	$
1	6	A$
2	4	ACA$
3	1	ATTACA$
4	5	CA$
5	0	GATTACA$
6	3	TACA$
7	2	TTACA$

index	off	sequence
0	7	$GATTAC*A*
1	6	A$GATTA*C*
2	4	ACA$GAT*T*
3	1	ATTACA$*G*
4	5	CA$GATT*A*
5	0	GATTACA*£*
6	3	TACA$GA*T*
7	2	TTACA$G*A*

The suffix array is a deceptively simple index consisting of the lexicographically sorted list of all suffixes in a genome sequence. Despite its simplicity, the suffix array is extremely powerful to accelerate sequence alignment computations, as it enables fast, variable-length lookups for any substring in the reference genome. Given a suffix array of a genome of G nt, searching for the location of any query requires at most $O(lg(G))$ probes to the array using a binary search (at most 32 probes for querying the human genome). In addition to these exact matching methods, suffix arrays also support inexact alignment algorithms that allow for some differences between the reference and query sequences. These applications commonly use a seed-and-extend technique that first finds relatively short exact matches using the suffix array to anchor the search for longer potentially in-exact matches. For a full description of the suffix arrays and their applications see Gusfield's classic text of sequence analysis[5].

Table 1 (left) shows the suffix array for the 7 nt string GATTACA$. In the computation, we include a special terminal '$' character to indicate the end of the string, and consider it lexicographically less than any other character in the string. The index field is the offset into the table, the SA field records the starting position of each suffix in the original string, and the suffix field shows the associated suffix. Explicitly storing each suffix has an intractable $O(G^2)$ space requirement, which is on the order of 10^{18} nt for the human genome. Therefore, in practice the suffix array records just the list of suffix offsets, and uses those offsets to lookup suffixes from a single copy of the full reference sequence. Here the SA ($[7, 6, 4, 1, 5, 0, 3, 2]$) records that the suffix starting at position 7 is the lexicographically smallest suffix, followed by the suffix starting at 6, and so forth. In this way, applications using suffix arrays require 15 GB for the human genome using 3 billion 4 byte integers to store the offsets and 3 GB to store the genome itself.

The closely related BWT index structure reduces the space requirement by using a permutation of the sequence as an index. The space needed for the BWT is therefore the same as the reference sequence itself, 3 GB for the human genome. The order of the permutation is determined by the suffix array. More precisely the BWT is the last column of the

[1] http://hadoop.apache.org

[2] http://www.genome10k.org

Burrows-Wheeler Matrix (BWM), which is a lexicographically sorted matrix of all of the cyclic permutations of the string. For example Table 1 (right) shows the BWM for the string `GATTACA$`, and the resulting BWT is `ACTGA$TA`. Note by construction, the first character of the BWT is always the last character of the string before the terminal '$' character. Furthermore, the i'th character of the BWT is the character preceding the i'th suffix in the suffix array ($BWT[i] = ref[SA[i] - 1]$), so that it is trivial to compute the BWT given the suffix array. Incredibly the BWT is by itself reversible using the Last-First property [2] which describes the implicit mapping between characters in the first and last column of the BWM. Using this property, the BWT is an implicit, space efficient encoding of the suffix array, and as such, nearly all of the new sequence alignment algorithms use the BWT as its index structure [11, 12] .

The leading algorithm for constructing the suffix array or BWT is the linear time DC3 difference cover algorithm [7]. A difference cover D modulo v is a subset of [0,v-1] such that for all i ∈ [0,v-1], there exists j,k ∈ D with i ≡ k-j(mod v). The most useful property of a difference cover is that it ensures a rank availability within length v. The basic idea of the algorithm is to divide the input into two parts of size two thirds and one third respectively. The larger part is sorted recursively and the result is used to sort the smaller third non-recursively in linear time. The two parts are then merged to give the final suffix array.

2.2 MapReduce

MapReduce was developed by Google for their large data analysis, especially for scanning trillions of webpages to compute the most relevant pages for a search query. The power of MapReduce is it can intelligently distribute computation across a cluster with hundreds or thousands of computers, each analyzing a portion of the dataset stored locally on the compute node. After an initial round of independent parallel computation, the machines efficiently exchange intermediate results, from which the final results are computed in parallel. The scale of MapReduce is substantial, and it has been reported that Google currently uses MapReduce to analyze nearly 1 exabyte of data every month. The system was originally developed for text and web processing and only available at Google, but an open source implementation called Hadoop is now available to install on any cluster for any application domain. Hadoop/MapReduce has provided researchers a powerful tool for tackling large-data problems in areas of machine learning [16], text processing [4], and bioinformatics [10, 18], to name just a few.

Parallel computation in MapReduce is structured into three major phases called map, shuffle, and reduce (See Figure 1). The map phase scans the input dataset and emits key-value pairs representing some relevant information of the data tagged by the key. These key value pairs are then distributed and shuffled so that all values associated with a given key are collected into a single list. Finally these key-value lists are then processed by the reduce function to compute the final results. The canonical MapReduce example is to count in parallel the number of occurrences of each word in large text corpus. In this example, the map function scans the input text and emits key-value pairs using the words as the keys, and the number 1 as the values. Then all values associated with a given word are collected to a single list on a single

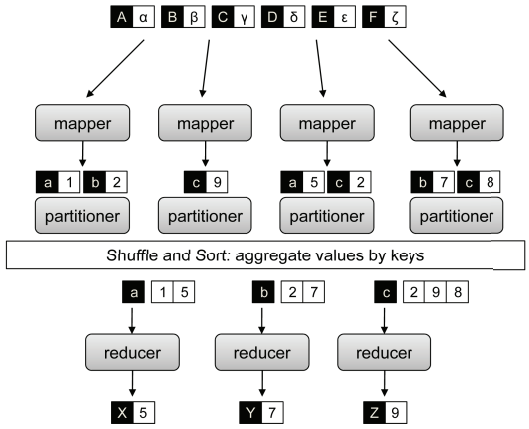

Figure 1: Illustration of MapReduce: mappers are applied to input records, which generate intermediate key-value pairs. The partitioner determines to which reducer the intermediate data are shuffled. The reducer then performs the final computation. Here the reducer sums the values for each key. (Figure derived from [13])

machine, so that the reduce function can sum the values to compute the total count of occurrences of each word.

The shuffle phase in Hadoop/MapReduce is implemented as a parallel distributed merge sort of the key-values pairs. As such, the reduce function will process the keys in sorted order and the final output from each reducer will naturally be sorted based on the key values. The subset of keys processed by a given reducer is determined by the partition function that bins the key-value pairs emitted by the map function. The default `HashPartitioner` function divides the set of keys into batches arbitrarily by computing a hash value of the key modulo the desired number of reducers. This is generally quite effective for balancing the load between reducers, but results in creating arbitrary subsets of keys for each reducer to evaluate. The Hadoop/MapReduce API also allows developers to override the default `HashPartitioner`, and use a customized partition function instead.

One particularly relevant custom partitioner was developed for the Gray sorting challenge[3]. The challenge evaluates the performance of various parallel systems sorting very large quantities of data. In the winning Hadoop entry, the map and reduce functions scan and emit the fixed length numerical records without modification. The critical insight was to use a custom `TotalOrderPartitioner` function that divides the space of keys into non-overlapping ranges based on distribution of a sample of the keys. This sampling is essential for maintaining proper load balance in the face of a non-uniform spread to the data. Once divided in this way, the values are then sorted by the parallel merge sort in the shuffle phase. Thus the final output will contain all of the input values globally sorted, although split into multiple but ordered files. Using such techniques, a Hadoop/MapReduce system won the Gray Daytona sort challenge in 2010, by sorting 1 trillion 100 byte records (100 TB) in 173 minutes using 3452 nodes.

The incredible performance of the parallel sorting methods developed for the Gray Daytona sort challenge suggests Hadoop could also be used to dramatically accelerate suf-

[3] `http://sortbenchmark.org`

fix array and BWT construction. The major difference is while the Gray Daytona challenge sorted short binary values, suffix array/BWT construction requires sorting variable length strings, up to the full length of the reference string. Adapting their algorithm for suffix array/BWT construction requires careful consideration, as a naive adaptation which explicitly generates every suffix for sorting would require an intractable amount of space for genome as long as the human genome. Furthermore, two strings of length n may require $O(n)$ time to compare, while the short fixed length values require $O(1)$ time.

3. BASIC INDEX CONSTRUCTION

The basic technique for a MapReduce implementation of suffix array and BWT construction is to use MapReduce to partition the suffixes into non-overlapping batches with lexicographically similar values, and then sort the batches of suffixes on different machines across the cluster. Much of this computation is embarrassingly parallel: suffixes can be independently assigned to partitions in parallel according to their prefix, and once formed, batches of suffixes with lexicographically similar values can be independently sorted. In between the batch identification and batch sorting, the algorithm needs to aggregate the suffixes into batches from all machines in the cluster, but this can be accomplished using the shuffle phase. As explained above, it is not tractable to distribute or sort the suffixes explicitly, but the algorithm can instead implicitly compare indexes after distributing the reference string to all the machines. In this way, there is a natural, if slow, implementation of suffix array construction in the MapReduce programming model:

0. Input The algorithm prepares a file with every suffix index in the reference string, i.e. 1,2,...,G where G is the length of the reference. The algorithm also distributes the reference string to every worker machine.

1. Map The mappers iterate over this file and emit key-value pairs with the prefix of the current suffix as the key, and the suffix index as the value. The prefix length is selected so that there are few suffixes assigned to a given batch. For example, using length 10 prefixes leads to $4^{10} = 1$ million possible batches, and each will contain on average 3000 suffixes for the human genome.

2. Shuffle The shuffle function aggregates all of the pairs with the same key and thus aggregates suffixes that begin with the same prefix.

3. Reduce The reducer sorts the suffixes within the batches using a string sorting algorithm, and outputs the suffix array or BWT.

This basic algorithm, while correct, requires considerable transient storage, especially since the input file will be at least $4G$ bytes and the mappers emit a total of $(P + 4)G$ bytes where P is the length of the prefix (commonly 8-15). These issues can be immediately improved as outlined in Figure 2. First, instead of creating a file with every suffix offset, the algorithm trivially preprocesses the reference string to determine it's length, and creates a file with independent ranges of suffixes for each mapper. This file contains M ranges, each spanning G/M of the reference, where M is the desired number of mappers. The algorithm uses the `NLineInputFormat` so that each line of the file (each range of positions) is considered a separate map task that can be executed in parallel. The second technique is to not explicitly record the prefix in the key-value pair, but to instead use a partitioner that can compute the partition value on-the-fly from the suffix index and reference string. A straightforward partitioning function of this type considers the length P substrings as a P digit number in base 5 and divides it by the desired number of reducers. Base 5 is used to encode the alphabet `A`=0, `C`=1, `G`=2, `T`=3, and also `N`=4 (meaning an ambiguous nucleotide). For example, the numerical value for the length 4 substring (also called a 4-mer) `ACGT` will be calculated as:

$$\text{mer} = 0 * 5^3 + 1 * 5^2 + 2 * 5 + 3 = 38_{10} \quad (1)$$

This partitioning scheme uniformly divides prefix mer values into independent ranges. For example if the number of reducers R is 10 and the mer length L is 5, the partitioner will assign $312 = (4 * 5^4 + 4 * 5^3 + 4 * 5^2 + 4 * 5 + 4)/10$ mers to each reducer. Thus mer values in the range 0-312 will be assigned to reducer 0, 313-614 to reducer 1, and so on. The reducer then collects all input indexes within a `SortedSet`, which sorts the indexes by comparing their associated suffixes using a copy of the reference string. The reducer then outputs the suffix array using the cleanup function after iterating over all of the indexes. The BWT can optionally be constructed instead by outputting the appropriate characters of the reference instead.

```
1: class Mapper
2:     method Map(key)
3:         (startIdx, endIdx) ← split(key, ',')
4:         for all idx ∈ startIdx : endIdx do
5:             Emit(idx)

1: class Partitioner
2:     method GetPartition(idx, value, n)
3:         substr ← dnaString.substr(idx, 15)
4:         return mer(substr)/numReducers

1: class Reducer
2:     method Reduce(id, valuex)
3:         sortedSet.add(id)
4:     method CleanUp(context)
5:         for all idx m ∈ sortedSet do
6:             Emit(idx)
```

Figure 2: Pseudocode for basic suffix array construction in MapReduce.

4. ADVANCED INDEX CONSTRUCTION

The above suffix array construction algorithm suffers two major problems when used for indexing large genomes that contain long repetitive sequences. First, the reference human genome sequence contains a region consisting of > 21 million `N`'s, representing a large region of ambiguity in the genome assembly. The above basic partitioner uniformly divides the space of mers, but there is a non-uniform distribution of mers in the genome: all 21 million of these suffixes have the same prefix value consisting of P N's, while most prefix mers occur just a few thousand times. Secondly, these repeats are very expensive to sort in the `SortedSet`, since many millions of characters will be examined to find the first

difference between them. This causes the runtime per reducer to approach $O(n^2\ log\ n)$ for repetitive regions of length n. In the following sections, we will describe our methods for addressing these major challenges: (1) to improve load balance among the reducers, we implemented a new partitioner that samples the genome to select the boundaries of the batches based on the true sequence distribution; and (2) to improve the batch sorting performance, we implemented an optimized recursive bucket sort that precomputes and determines problematic repeats on-the-fly.

4.1 Sampling Partitioner

The sampling partitioner uses dynamic ranges to balance the number of suffixes per batch, rather than the number of mers per batch as in the basic algorithm. The boundaries between batches are determined by $R-1$ variable length mers computed from a sorted list of suffixes uniformly sampled from the reference string, where R is the desired number of reducers. The variable length boundary mers enables long repetitive sequences, with correspondingly long identical prefixes, to be subdivided into separate batches. The boundary mers are recorded using a runlength encoding so that they contain at least 2 distinct characters. For example if there is a repeat region of 100,000 `N`'s followed by `G`, the algorithm may create boundary mers of the form `N:5000,G:1` and `N:100000,G:1`. These boundary mers are stored in a file and distributed to all of the partitioners at launch. Furthermore, the partitioner quickly computes the partition for each suffix by exploiting the relationship between consecutive suffixes. For example if the suffix starting at position a begins with a prefix of L `N`'s, the algorithm can immediately compute the suffix starting at position $a+1$ begins with a prefix of $L-1$ `N`'s, thus avoiding substantial unnecessary comparisons. Finally, the partition represents each character of the reference string with at most 3 bits, and reduces the memory requirement for analyzing the human genome from 3.0 GB to 1.2 GB.

4.2 Recursive Bucket Sorting

The reducers are responsible for the fine grained suffix sorting within the partitions. A basic Quicksort algorithm will approach $O(n^2\ log\ n)$ time for n suffixes, because it can $O(n)$ time to compare two suffixes. Instead our algorithm uses a recursive bucket sort (Figure 3). The algorithm first orders the suffixes based on a *plen* length prefix (initially $plen = 15$) using a version of Quicksort that only compares the first *plen* characters of a suffix. It then scans the list, and recursively sorts blocks of suffixes with the same prefix. To limit the recursion depth, if the size of the current range is less than $MinBucketSize$ (default: 10% of the original range), the algorithm reverts to a Quicksort over the entire suffix length. In practice, the recursive bucket sort is very effective for sorting most sets of suffixes because it restricts how many suffixes will be compared over their full length. However, it is not better than Quicksort for sets of suffixes that have very long identical prefixes. In large genomes this is especially problematic for two classes of repeats:

(1) Suffixes from long single character repeats
(2) Suffixes from long multiple character repeats

The following optimizations address these challenges and greatly accelerate the recursive sort. Even with the optimizations, the memory requirement for the reducers is proportional to the number of suffixes assigned to each reducer.

```
1: method BucketSort(indices, start, stop, plen)
2:     if stop − start < 2 then
3:         return
4:     if stop − start < MinBucketSize then
5:         QSort(indices, start, stop)
6:         return
7:     PrefixQSort(indices, start, stop, plen)
8:     split = start
9:     for i ∈ [start + 1 : end] do
10:        if PrefixCmp(i − 1, i, plen) > 0 then
11:            BucketSort(indices, split, i − 1, plen + 15)
12:            split = i
13:    BucketSort(indices, split, stop, plen + 15)
```

Figure 3: Pseudocode for recursive BucketSort.

Hence increasing the number of reducers decreases the number of suffixes per reducer, decreases the peak memory usage, and improves the runtime.

4.2.1 Optimizing single character repeats

Single character repeats, including both biologically meaningful mononucleotide repeats and also regions of ambiguity marked by `N`, are very common in large genomes. Since they can lead to quadratic behaviour while sorting, our algorithm scans the suffixes, and precomputes the length of the single character repeats in their prefixes. This computation is optimized using the fact that if suffix a begins with a single character repeat of length L, then for all $d < L$ suffix $a+d$ begins with a repeat of length $L-d$. The precomputed repeat lengths are then used in the sort routines to skip directly to the first non-repetitive characters to determine in $O(1)$ time which suffix from a given repeat is lexicographically less. In this way, a set of consecutive suffixes from a repeat can be sorted in linear rather than quadratic time. Our algorithm also uses this same logic when comparing suffixes from different repeats in the genome. For example, if the suffix starting at position a begins with a repeat of length L_a is compared to the suffix at position b that begins with a repeat of the same character of length L_b, the algorithm compares the lengths L_a to L_b, and then the characters starting at $a+L_a$ and $a+L_b$ without needing to compare the repetitive characters.

4.2.2 Optimizing multiple character repeats

Unlike single character repeats, it is considerably more difficult to precompute multiple character repeats, since this computation may need to consider all possible repeat lengths. In practice, repeats of this type are rarer than single character repeats, but common enough to cause noticeable slowdown for the reducers which sort them. To address this problem, our algorithm uses two techniques to record on-the-fly the length and composition of certain repeats as they are discovered. First, the algorithm uses the ranks of the previously sorted suffixes to accelerate further comparisons, similar to the difference cover algorithm. For example, if suffixes a and b were previously sorted, at most d characters are compared to evaluate the suffixes at $a-d$ and $b-d$. The second optimization is to maintain the maximum repeat length encountered along with the corresponding index pairs (even if the order of those index pairs has not yet been fully determined). If the comparison of two suffixes reaches those pairs, the algorithm skips the repetitive characters to proceed to the next differing character positions.

5. RESULTS

We implemented the above parallel suffix array and BWT construction algorithms for Hadoop 0.20.0. The mapper and partitioner were implement in Java, while the reducer was implemented in C++ (called via JNI) as this allowed more control and performance over the critical recursive sort routines. The reducers use shared memory via the Unix function `mmap()` to load a single copy of the reference genome for any number of reducers on the same machine. During the evaluation we executed our code on Hadoop clusters with 30, 60, or 120 concurrent tasks, using machines leased from the Amazon Elastic Compute Cloud (EC2) booting the virtual machine image `AMI-6AA34003`. We performed the experiments using High-Memory Double Extra Large (m2.2xlarge) instances, which provide 4 hyperthreaded cores at 3.2 "EC2 Compute Units" (roughly 3.2 GHz), 34.2 GB of RAM, and 850 GB of storage. For the 30 and 60 core evaluations, we used 10 and 20 nodes with 3 tasks per node; for the 120 core evaluation we used 20 nodes with 6 tasks per node. These machine types were selected because they are the least expensive instance types with sufficient RAM for storing the entire suffix array in memory as would be necessary for using the suffix array after construction. At the time of the evaluation, these instances cost $1.00 per hour per node. Each evaluation finished well within 1 hour, so each evaluation cost at most $21 including the cost for the Hadoop master node.

For the evaluation, we constructed the suffix array of five important genomes ranging in size from 370 million to 3.1 billion nucleotides including 3 mammals, 1 bird, and 1 plant genomes (Table 2). The rice genome OS1 was downloaded from NCBI[4] and the others were downloaded from UCSC[5]. The human genome (`HG19`) is the largest genome available to date, but as explained above, there are organisms with (substantially) larger genomes that have not yet been sequenced for which our methods will be even more important. For example, efforts are currently underway to sequence and assemble the 24 billion nt loblolly pine (*Pinus taeda*) genome.

Figure 4 shows the end-to-end runtime for constructing the suffix arrays in the 3 cluster environments. The figure shows the performance greatly improved for as many as 120 processing cores, although the large genomes demonstrate better scaling to large numbers of cores. For example, the suffix array construction for `HG19` using 60 cores was 1.96x faster than the 30 core cluster, and the 120 core cluster was 2.37x faster than the 30 core cluster. In `OS1`, the 60 core cluster was 1.45x faster than the 30 core cluster and the 120 core cluster was only 1.53x faster.

We investigated the cause of non-linear speedup and concluded our algorithm was substantially limited by the Hadoop overhead. In particular, on the 120 core cluster for `HG19`, 120 s are spent writing the suffix array to the HDFS using the default 3-fold replication. Furthermore, it required 398 s for the 120 core cluster to processes `HG19` using the default `HashPartitioner` and an Identity Reducer that simply writes the unsorted suffix indexes to the HDFS. This suggest approximately half of the runtime is spent in the overhead of launching Hadoop, generating and shuffling the suffix indexes, and then writing the (unsorted) suffixes to the HDFS. Figure 5 shows the average runtime required just for the reducers while executing the experiments in Figure 4. In this experiment, a timer was started after the set of suffix indexes was collected by the reducer and stopped after the sorted indexes were written to the local disk. This shows the reducers (in isolation) had better scaling properties than the overall end-to-end runtime: in `HG19`, the 120 core cluster was 2.86x faster than the 30 core cluster, and in `OS1`, the 120 core cluster was 3.13x faster than the 30 core cluster.

Figure 6 plots the running time as a function of the genome size for the 5 test genomes as 3 cluster environments. For a given cluster size, there is an approximately linear relationship between the genome size and the runtime, indicating that our overall algorithm performs well in the presence of complicated repeats. Note without the enhancements outlined in Section 4, the running time is non-linear, and `HG19` requires more than over 1 hour for the 60 core cluster (data not shown). As expected the slope of the relationship (seconds/nt) decreases as the cluster size increases.

The final evaluation was to compare the results of our new parallel algorithm against the leading serial and parallel methods for suffix array and BWT construction. For the suffix array construction, we evaluated against the widely used

Table 2: Genomes evaluated.

Name	Genome	Build	Length (nt)
`HG19`	Human (*Homo sapiens*)	19	3,095,677,412
`MM9`	Mouse (*Mus musculus*)	9	2,654,895,218
`BT4`	Cow (*Bos taurus*)	4	2,634,413,324
`GG3`	Chicken (*Gallus gallus*)	3	1,031,883,471
`OS1`	Rice (*Oryza sativa*)	1	370,792,118

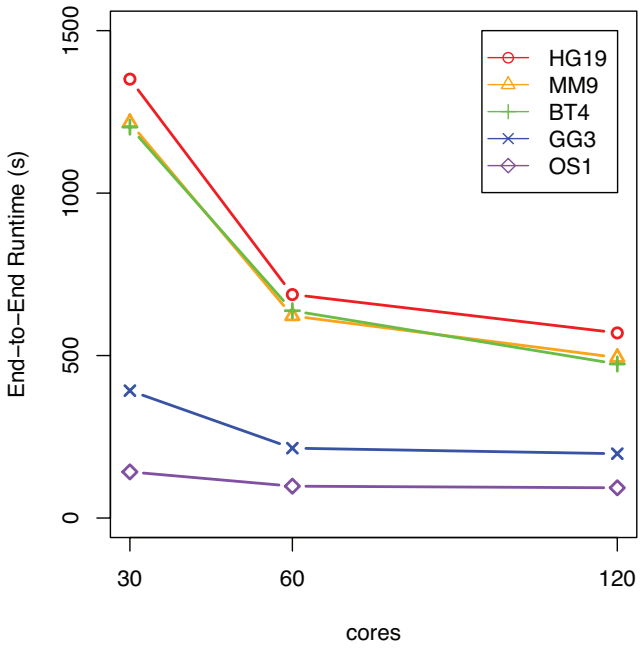

Figure 4: End-to-End Suffix Array Construction Runtime. All genomes show substantial improvement in runtime between the 30 and the 60 core cluster. The larger genomes continue to improve on the 120 core cluster, while the smaller genomes are limited by overhead at this level of parallelism.

[4] `ftp://ftp.ncbi.nih.gov/genomes/`

[5] `ftp://hgdownload.cse.ucsc.edu/goldenPath/`

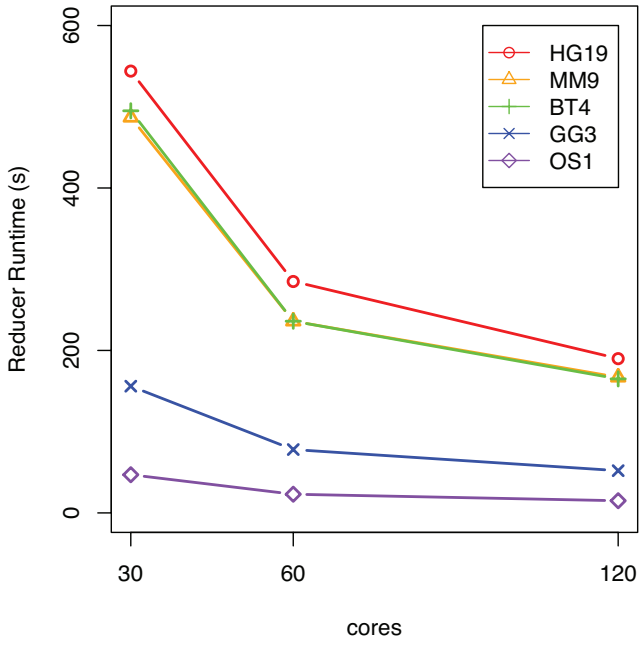

Figure 5: Average Reducer Runtime. The average runtime of the reducers shows improved scaling performance compared to the end-to-end runtime.

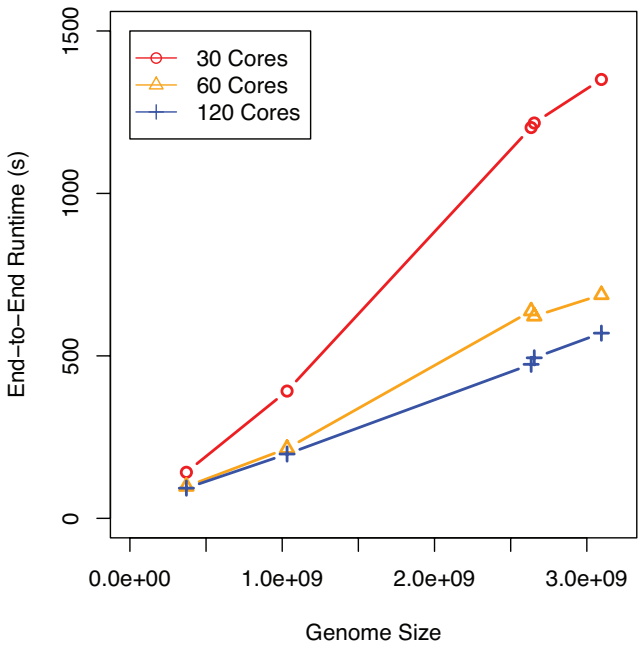

Figure 6: Runtime versus Genome Size. For a given number of cores, the end-to-end runtime is approximately linear across the 5 genome sizes.

application Vmatch[6] (version 2.0), which internally uses a highly optimized multikey Quicksort algorithm similar to our implementation. For BWT construction we evaluated against the leading application Bowtie [11], which internally uses the difference cover algorithm. We considered two scenarios in the evaluation: constructing the indexes using a desktop workstation using 7GB of RAM (on an Opteron 250 running at 2.4GHz), and a massive server with 512GB of RAM (on an Intel Xeon E7540 running at 2.0GHz). The large memory server was considered in the evaluation because Vmatch requires 49.1GB of RAM for `HG19` and is thus out of reach for many users, whereas Bowtie can execute on a regular desktop workstation at the expense of speed. Table 3 shows the end-to-end speedup in the two environments. Bowtie computes the BWT of both the genome and then the reverse of the genome without any option to compute just one direction, so here we report half the running time of Bowtie when computing the speedup.

The end-to-end speedup was modest compared to the number of cores used in the experiments, but nevertheless reduced the runtime from hours to minutes represents a significant advance for researchers that regularly need to construct these indices. Furthermore, the fastest published suffix array construction of the human genome was computed using the PDC3 [8] algorithm in 181 s using 128 2.0GHz Intel Xeon processors. This is three times faster than our implementation using 120 cores, but their evaluation required a specialized 800 Mbyte/s Quadrics QSNet II network with an order of magnitude higher bandwidth than commodity Gigabit Ethernet. Our code is likely to perform better using such specialized network equipment as well. They also did not consider the time to distribute the genome sequence to the worker nodes in their evaluation.

[6] `http://www.vmatch.de/`

Table 3: Speedup compared to Vmatch and Bowtie in server and desktop environments, respectively.

			Speedup		
Name	Algorithm	Time (s)	30C	60C	120C
HG19	Vmatch	4475	3.31x	6.50x	7.85x
	Bowtie	7206	5.33x	10.47x	12.64x
MM9	Vmatch	4494	3.69x	7.23x	9.10x
	Bowtie	6632	5.45x	10.66x	13.43x
BT4	Vmatch	4124	3.43x	6.46x	8.70x
	Bowtie	7075	5.88x	11.09x	14.93x
GG3	Vmatch	1218	3.11x	5.67x	6.15x
	Bowtie	1878	4.79x	8.73x	9.48x
OS1	Vmatch	421	2.96x	4.30x	4.53x
	Bowtie	728	5.13x	7.43x	7.83x

6. DISCUSSION

Here we have presented a novel open-source parallel algorithm for the important task of indexing a genome using a suffix array or the Burrows-Wheeler Transform. This computation is essential for enabling almost all downstream analyses of a genome, and is therefore computed for virtually every sequenced organism, including for the 10,000 vertebrate genomes planned to be sequenced as part of the Genome 10k project. Since it is often necessary to index the intermediate genome assembly multiple times, this project alone will spend >5 CPU years indexing genomes using the current leading methods, and thus using our methods instead will be a very significant improvement.

The essence of our algorithm is to use MapReduce to partition the suffixes into non-overlapping batches of lexicographically related sequences, and then independently sort each batch across the cluster. This computation lends itself well to the MapReduce programming model, but several signif-

icant algorithmic insights were necessary to make the algorithm practical for analyzing the suffixes of large genomes. Critical to the performance of our algorithm was to implement several techniques for exploiting the interdependencies between repetitive suffixes. While these techniques do not guarantee linear time performance for all sequence compositions, in practice do exhibit nearly linear performance across our collection of 5 very distantly related organisms.

In experiments with up to 120 cores, the algorithm leverages the unique large-scale data processing capabilities of Hadoop to reduce the runtime of this important computation from several hours on a desktop workstation to mere minutes on the Amazon Elastic Compute Cloud (EC2). Furthermore, since the experiments were performed on the Amazon EC2, any researcher in the world can take advantage of our advance. In contrast, the MPI-based PDC3 is able to compute the suffix array of the human genome faster with similar numbers of cores, but only when used with a speciality networking hardware not available to most researchers.

The end-to-end speedup was relatively modest compared to the number of cores, and future work remains to further refine the implementation. A significant fraction of the runtime is for the Hadoop overhead, so improvements to the core Hadoop subsystems will likely have the most impact on end-to-end performance. Nevertheless, our algorithm improves the wall clock performance from several hours to less than 10 minutes for the mammalian genomes and thus represents a very significant advance. Other future work remains to extend the algorithm to compute other sequence analysis algorithms in parallel. For example, several new sequence error correction algorithms are based on constructing and analyzing the suffix array of a large set of sequencing reads [6]. Because of the scale of the data involved, these methods have not yet been used For mammalian genomes, but our methods suggest it may be possible to successfully scale up these algorithms as well.

7. ACKNOWLEDGEMENTS

The authors would like to thank Steve Skiena at Stony Brook University, Adam Phillippy at NBACC, and Ben Langmead at the University of Maryland for their helpful discussions. We also thank Amazon Web Services for providing credits for testing under the AWS in Education Research program. This work was supported, in part, by the Simons Center of Quantitative Biology at Cold Spring Harbor Laboratory.

8. REFERENCES

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[2] M. Burrows and D. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Technical report, 1994.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, pages 137–150, San Francisco, California, 2004.

[4] C. Dyer, A. Cordova, A. Mont, and J. Lin. Fast, easy, and cheap: Construction of statistical machine translation models with MapReduce. In *Proceedings of the Third Workshop on Statistical Machine Translation at ACL 2008*, pages 199–207, Columbus, Ohio, 2008.

[5] D. Gusfield. *Algorithms on strings, trees, and sequences : computer science and computational biology*. Cambridge Univ. Press, January 2007.

[6] L. Ilie, F. Fazayeli, and S. Ilie. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, February 2011.

[7] J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53:918–936, November 2006.

[8] F. Kulla and P. Sanders. Scalable Parallel Suffix Array Construction. pages 543–546. 2007.

[9] S. Kurtz, A. Phillippy, A. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. Salzberg. Versatile and open software for comparing large genomes. *Genome Biology*, 5(2):R12+, 2004.

[10] B. Langmead, M. C. Schatz, J. Lin, M. Pop, and S. L. Salzberg. Searching for SNPs with cloud computing. *Genome Biology*, 10(R134), 2009.

[11] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25+, 2009.

[12] H. Li and R. Durbin. Fast and accurate long-read alignment with BurrowsâĂŞWheeler transform. *Bioinformatics*, 26(5):589–595, March 2010.

[13] J. Lin and M. Schatz. Design patterns for efficient graph algorithms in MapReduce. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[14] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[15] E. R. Mardis. Next-generation DNA sequencing methods. *Annual review of genomics and human genetics*, 9(1):387–402, June 2008.

[16] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo. PLANET: Massively parallel learning of tree ensembles with MapReduce. In *Proceedings of the 35th International Conference on Very Large Data Base (VLDB 2009)*, pages 1426–1437, Lyon, France, 2009.

[17] S. J. Puglisi, W. F. Smyth, and A. Turpin. The Performance of Linear Time Suffix Sorting Algorithms. In *Data Compression Conference*, pages 358–367, Washington, DC, USA, 2005. IEEE.

[18] M. C. Schatz. CloudBurst: Highly sensitive read mapping with MapReduce. *Bioinformatics*, 25(11):1363–1369, 2009.

[19] M. C. Schatz, A. L. Delcher, and S. L. Salzberg. Assembly of large genomes using second-generation sequencing. *Genome Research*, 20(9):1165–1173, September 2010.

[20] C. Trapnell, L. Pachter, and S. L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, May 2009.