DETECTING GENOMIC ELEMENTS OF EXTREME CONSERVATION IN HIGHER EUKARYOTES BY INTEGRATION OF HASH MAPPING AND CACHE-OBLIVIOUS IN-MEMORY COMPUTING

A Thesis presented to

the Faculty of the Graduate School at the

University of Missouri

In Partial Fulfillment

of the Requirements for the Degree

Masters of Science

by

ANDI DHROSO

Dr. Dmitry Korkin, Thesis Supervisor

May 2015

The undersigned, appointed by the Dean of the Graduate School,

have examined the thesis entitled:

DETECTING GENOMIC ELEMENTS OF EXTREME CONSERVATION IN HIGHER EUKARYOTES BY INTEGRATION OF HASH MAPPING AND CACHE-OBLIVIOUS IN-MEMORY COMPUTING

presented by Andi Dhroso

A candidate for the degree of

Masters of Science

and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Dmitry Korkin

Dr. Dong Xu

Dr. Shyu

Dr. Conant

DEDICATION

Dedicated to my beautiful wife whom without I would be lost.

Thank you for all your continuous support and motivation,

unconditional love and for always believing in me.

ACKNOWLEDGEMENTS

First I would like to thank my thesis advisor Dmitry Korkin for all his support, motivation and guidance throughout this project. Would like to thank Gavin Conant and Chris Pires for helping me with all the biological insights, Chi-Ren Shyu as this work was inspired by his previous work, and all the committee members for giving me the opportunity to present my research work.

Also, thanks to Mike Phinney and Hongfei Cao for all the discussions and my fellow labmates for their continuous support. Special thanks goes to Blake Anderson for providing invaluable insights toward the project and many discussions.

Last but not least, I am forever grateful to my family for their continuous support as without their sacrifice I would not be the person that I am today.

Table of Content

List of Tables

List of Figures

DETECTING GENOMIC ELEMENTS OF EXTREME CONSERVATION IN HIGHER EUKARYOTES BY INTEGRATION OF HASH MAPPING AND CACHE-OBLIVIOUS IN-MEMORY COMPUTING

Andi Dhroso
Dr. Dmitry Korkin, Thesis Supervisor

Abstract

Genomics is one of the first life science disciplines to enter the era of Big Data, facing challenges in all three dimensions—volume, variety, and velocity. Yet, in spite of a plethora of sequencing data, we are still far from creating a complete encyclopedia of functional and structural elements of the genome. In 2004, an example of this knowledge gap came about when Bejerano and Haussler discovered 481 DNA elements in the syntenic positions of human, mouse and rat genomes that were 100% identical, called the ultra-conserved elements (UCEs). Recently, using an advanced data-mining alignment-free approach, it was shown that this phenomenon exists beyond the animal kingdom and outside the regions of synteny (conservation of blocks of order within two sets of chromosomes that are being compared with each other).

Our ultimate goal is to provide a comprehensive atlas of the regions of extreme conservation in higher eukaryotes providing insights into the structural organization, function and evolution of these elements. However, the all-against-all comparison of dozens, if not hundreds of eukaryotic genomes may not be feasible using current approaches. For instance, the original findings of syntenic-only UCEs relied on a whole-genome alignment of three mammalian genomes and it took one day on a 24-nodes cluster. A comprehensive alignment-free algorithm that guaranteed finding all syntenic and non-syntenic long identical multi-species elements (LIMEs) took three days on a 48 CPU cluster between two assembled genomes.

Here, we present a new hybrid approach that integrates the ideas of hash mapping and cache-oblivious in-memory computing. Our algorithm leverages the concept of help-me-help-you, where the data structures are tailored to maximize cache-hit while minimizing cache-miss. As a result, our hybrid algorithm is approximately 800 times faster than the current state-of-the-art method and is scalable to deal with the unassembled genomes. The new hybrid approach has been applied to detect the earliest evidence of extreme conservation by including into the large-scale analysis recently sequenced genomes of coelacanth and lamprey. The integration of efficient software with hardware-optimized approaches has shown to be a promising direction in comparative genomics, allowing scientists to provide even deeper insights into the function and evolution of eukaryotic genomes.

CHAPTER 1

Introduction

1.1 Overview

DNA is the genetic material for all life on earth. It contains all of the necessary information and instructions for species in order to survive. One of the questions that has puzzled scientists since the dawn of time is the “Where did life begin or even, How did it begin?” Thanks to technological breakthroughs and enormous amount of scientific efforts, we have greatly increased our knowledge about our genetic material. Ever since the human genome project has been complete [1, 2] we have been able to identify locations of many human genes and provided information regarding the structure and organization. Many organisms have been sequenced, allowing scientists to build a repertoire which can then facilitate and lead to knowledge discovery. Here in our work, we introduce a hash-mapping, alignment-free information retrieval method, in which detects LIMEs across assembled and un-assembled genomes (at the chromosome and contig level respectively).

The topic of Big Data has become more and more relevant in the field of genomics [1]. A little over a decade ago was when we were only able to sequence one’s genome only at depths of 27 million reads, 5-folds the genome size [2] where cost was significant [3] **Figure 1.0**. Thanks to significant advances in technology in the recent decade, now we are able to sequence genomes at significantly much more deeper level (over 200 million reads), and manageable cost. Moreover, having the ability to sequence genomes at much deeper levels allows scientists to provide even deeper insights into the function and evolution of eukaryotic genomes. However, this implies that data generation from the sequencer is also significantly much larger than in the past; as a consequence, the demand for higher storage capacity has grown much more. Besides the increase in data generation, the rate in which data is being generated has also

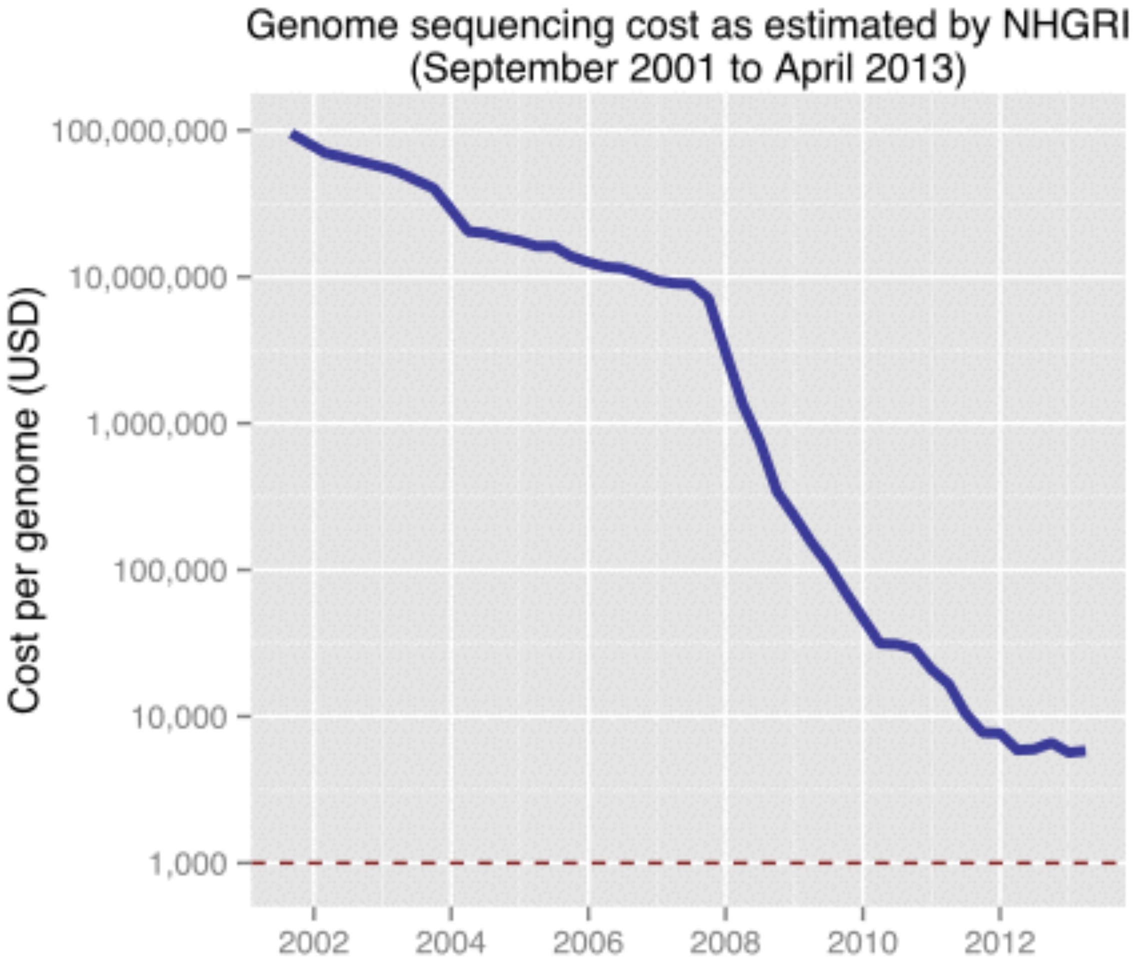

Figure 1.0: Estimated cost of genome sequencing

increased tremendously, demanding more and more increase in computational power. This phenomena is observed not only in the field of genetics, but in many others as well [4]. Such fields include: Physics, Image Processing (satellite high definition imagery…etc), and Telecommunications (surveillance in audio and video domain)…etc. Now more than ever, we face challenges in all three dimensions — data that needs to be processed with respect to volume, complexity, and/or velocity. Although to this day categorizing data as Big Data is not clearly defined, many in the scientific community will agree that Big Data is computationally demanding, thus requiring not only better hardware, i.e. faster processors and large amount of memory resources, it also requires us to re-think the way in which we represent our data; furthermore, this also means when we process data, we only use the minimum set of data in order to process and retrieve the results in a practical and feasible time frame. Losing generality or being specific with respect to data may seem unorthodox in the field of Computer Science, however, it is necessary in order for us to obtain our results in a feasible amount of time. In addition, having such large scale and complex genomic datasets, we can no longer apply the one-size fits all concept as we were able to do at some point not too long in the past. Consequently, this means that we can no longer apply some of the same algorithms and/or data structures in a generic form and expect to achieve optimum performance or the same performance as when the algorithms were first benchmarked - since we no longer deal with the same scale of data. This is particularly important especially in the field of Genomics as we can provide and achieve faster results, resulting in faster disease cure and an overall better understanding of our genetic properties as well as function. Thus, in order to process such data, we have to optimize our algorithms not only at the software level, we also must organize our data in such a fashion in order to take advantage of the software and hardware optimizations as well. We refer to this approach as in-memory computing help-me-help-you concept.

1.2 Current Genomic DNA Sequence Searches

From the evolutionary point of view, identifying the origins in which the species evolved several components can be considered, i.e. genetic makeup such as their DNA sequence, function…etc. In addition, having the ability to measure species' similarity provides interesting insights as to how the species evolve with respect to clade, *i.e.* jawless fish such as lamprey to jawed fish such as elephant shark to as recent as the tetrapods. One question in particular we can ask, is, "are there any particular functions or sequences that get preserved (or are immune to mutations) over millions of years that are necessary for the survival of species. Moreover, what changes came about the genetic material (whether added or removed functions) in order for species to evolve from being aquatic to land type animals.

Significant number of methods have been developed to detect sequence similarity, substring matching…etc [5-14] to help scientists further understand how our genetic material is organized and functions. Many of these methods however, rely on sequence alignment, consequently cannot not provide the entire comprehensive set of conserved elements. For example genes that have incurred to transposition cannot be detected using sequence alignments approaches. On the contrary, Ning *et al* 2001 [14], developed a method incorporating a hash-mapping and alignment-free sequence search approach. However, because it keeps the hash lookup-tables in memory for all species, one of the issues with large dataset is scalability. Stokes *et* al 2006 [15] also developed a search algorithm by generating overlapping k-mers and storing them into a hash table as an intermediate placeholder to speed up retrieval, however the algorithm search turns into a brute force (naive approach) if there is too much degradation when query size is small and the hash-map lookup-table is large. Another hash-mapping algorithm is FLASH [34]. Flash takes the noncontiguous DNA search-keys, concatenates and indexes them. Unfortunately, the number of concatenated strings becomes very large, where scalability for multiple genomes becomes impractical. The current state-of-the-art approach [16], utilizes some of the same concepts of hash-mapping, significantly improving performance and scalability when processing multiple genomes. In addition, it is an alignment-free method, consequently finding LIMEs in syntenic and non-syntenic regions. Furthermore, in contrast to its superior performance compared to other previous methods, sequences assembled up to the contig and/or scaffold level becomes impractical.

1.3 Ultra-Conservative Elements (UCEs)

In 2004, there were identified 481 ultra conserved elements (UCEs) that are absolutely identical (100% identical with no insertion, or deletions) with length longer than 200 base pairs (bp) between orthologous regions of the human, mouse and rat [17]. As much as 5% of the human genome was estimated to be more conserved as expected from the neutral evolution model from the time since the split between human and rodents, thus, implying negative or purifying selection [18-20]. Using the slowest neutral substitution rate, the probability of finding even one such element in approximately 3 billion bases is 10^{-22}, suggesting that the behavior of such conservation "exhibited in the ultra conserved elements must result from the onset during chordate evolution of either a highly elevated negative selection rate in these regions (about a 20 times smaller chance of mutations becoming fixed in the population), a highly reduced mutation rate (about 20 times fewer mutations), or some combination of these effects" [17]. This suggestion was also concluded in Stephen *et al* 2008 work by investigating several placental mammals - human, opossum, chicken, and frog to name a few. Employing a multiple sequence alignment method, Stephen *et al* 2008 [21] identified 13,736 UCEs of length at least 100 bp and 2,189 sequences over at least 200 bp, greatly expanding the UCEs repertoire [21]. Sequence identity between the human and chicken was approximately 96% similar, implying an extremely low substation rate in UCEs, 1% per site per 100 million years. If such low substitution rate were to remain constant through the tetrapods, we should be able to find the majority if not all of the UCEs that are present in the tetrapods, to be present in fish. However, this is not the case in Bejerano *et al* 2004 [17] work, suggesting, first, that the substitution mutation rate within the UCEs has changed throughout evolution, and second, such phenomena could be explained due to function change required in the amniote, tetrapods or both.

Stephen *et al* 2008 [21], identified 14,000 UCEs of length greater than 100 base pairs that were identical in at least 3 out of 5 species in which were identified as to have appeared during the tetrapods era, existing with a significant molecular slowdown in their molecular clock, particularly amniotes and tetrapods, demonstrating a genome-wide function adaptation, followed by purifying selection. In Stephen *et al* 2008 [21], authors analyzed Human-centric and mouse-centric multiple alignments of 17 species (human hg18, mouse mm8, rat rn4, dog canFam2, cow bosTau2, opossum monDom4, chicken galGal2, frog xenTro1, puffer fish fr) from the University of California Santa Cruz (UCSC) genome browser (Kuhn *et al*. 2007), in which excluded unassembled, haplo-type and mitochondrial chromosomes. The identical regions were scanned in 3 placental mammals. Between human-dog-cow there were 11,110 identical elements with length of 100 base pairs or longer, 5,505 elements between human-mouse-rat, and 5,546 elements between mouse-rat-dog. In addition, zebrafish-centric multiple alignments of seven species (zebrafish danRer4, tetraodon tetNig1, and puffer fish fr1) and as well as stickleback-centric multiple alignment of 8 species stickleback (February 2006, gasAcu1), medaka (October 2005, oryLat1), puffer fish (October 2004, fr2), tetraodon (February 2004, tetNig1), zebrafish (March 2006, danRer4), chicken (May 2006, galGal3), mouse (February 2006, mm8), and human (March 2006, hg18) were analyzed as well - identifying 43 elements of having length of 100 bp or longer between the three fish. By taking the union of elements in all three sets: human-dog-cow, human-mouse-rat and mouse-rat-dog, and then mapping these elements in human coordinates resulted in 13,736 eutherian UCEs of 100 base-pairs or longer - covering 2.1 mega-bases, in which 2,189 of them were at least 200 base-pairs. Stephan *et al* 2008, went a step further analyzing the birth and death of these ultra conserved elements by dividing them into groups based on when the UCEs were first detected in evolutionary history with major increase in elements appearing before tetrapods speciation. Approximately 5,500 (about 40%) ancient elements were in fish, 4,160 elements (about 30%) were first detected in frog and 2,540 (18%) first appeared in the amniotes. Further analysis showed that the UCEs are located near or within genes that are involved in regulation of transcription and of development [29,30,31].

Following Bejerano *et al* 2004 work [17], Reneker *et al* 2012 [16], introduced a new computational approach, in that it is alignment-free, whereas alignment based method were utilized in [17, 21]. In addition to 6 mammalian species (human, mouse, rat, macaque, chicken, and dog) Reneker *et al* 2012 [16] analyzed also 6 plant species: (arabidopsis, soybean, rice, cottonwood, sorghum, and grape) in which were analyzed as well. Ultra conserved elements that lied in syntenic regions as well as elements that were not in non syntenic regions were identified. The comparative analysis showed that in contrast to mammalian genomes where there were syntenic and non-syntenic elements, no LIMEs were detected in syntenic regions in plants. Additionally, complex (non-repetitive) LIMEs were found to be near each other if not overlapping. Approximately 92% of the complex limes found were found to be overlapping. There were also found only 241 distinct motifs that made up the repetitive LIMEs, ranging from 2 to 30-bp. Similar to Stephen *et al* 2008 [21], Reneker *et al* 2012 [16] found the entire set of UCEs (481 that were found initially by Bejerano *et al* 2004 [16]) plus novel ones, where other alignment methods were not able to detect. The distribution of LIMEs showed multiple copies of a single LIME within one chromosome as well as multiple chromosomes [16, 21]. It was hypothesized the reason for LIMEs to be found in syntenic and non-syntenic regions in animals and only in non-syntenic regions in plant is because in plants, lack of synteny is attributable to the elements having been “created in place” rather than inherited from the common ancestor. Reneker *et al* 2012 identified 503 unique complex with length greater than 200 base pairs, however, this number increases to 619 if subsequences that map to distinct locations in at least one genome, in comparison to 481 elements found by Bejerano *et al* 2004.

In this work, we have developed a hash-map, alignment free method to identify LIMEs not only on assembled genomes, but unassembled genomes as well. We demonstrate that our method is able to scale and able to process large number of genomes, i.e. assembled or un-assembled, something that to this date, no known methods are able to perform such processing in a feasible amount time. We have done this by utilizing the help-me-help-you concept. By optimizing our data structure we reduce the amount of processing time at the smallest operation. Thus, we are able to recall the full coverage and in a fraction of the time that was reported by the current state-of-the-art method.

CHAPTER 2

Methodology

2.1 Substring matching problem

The problem of exact string matching can be formulated as the problem of finding all longest common substrings between two or more sequences [22]. Assume sequences are strings of characters defined over a finite alphabet Σ, then let a and b be two sequences of lengths n and k respectively. For sequence $a = s_1s_2\ldots s_n$, sequence $b = s_{i1}s_{i2}\ldots s_{ik}$ is called a substring of **a** if $1 \leq i_j \leq n$, for $1 \leq j \leq k$, and $i_s < i_t$, for $1 \leq s < t \leq k$. Let $S = \{a_1, a_2, \ldots, a_d\}$ be a set of sequences over alphabet Σ. A multiple longest common substring for set S is a sequence b such that (i) b is a substring of $a_i, 1 \leq i \leq d$, and (ii) b is the longest one satisfying (i). Typically methods for the multiple longest common substring are based on dynamic programming [23, 24]. One of the attractive components of dynamic programming is its simplicity. Given two sequences a_1 and a_2 of length n_1 and n_2 respectively, a dynamic programming algorithm iteratively constructs a $n_1 \times n_2$ score matrix L, in which $L[\,i,j\,]$, $0 \leq i \leq n_1$, $0 \leq j \leq n_2$ is the length of a longest common substring between two prefixes $a_1[1, \ldots, i\,]$ and $a_2[1, \ldots, j\,]$.

$$L[i,j] = \begin{cases} 0, & if\ i\ or\ j = 0 \\ L[i-1, j-1]\ +1, & if\ a_1[i] = a_2[j] \\ max(L[i, j-1], L[i-1, j]), & if a_1[i] = a_2[j] \end{cases} \tag{1}$$

However, despite of its straight forward implementation, dynamic programing becomes impractical when dealing with large dataset such as genomic data. Dynamic programing calculates all entries in L, thus the resulting algorithm incurs a time and space complexity of *$O(n^d)$* for *d* sequences of length *n* [4].

Finding unknown substrings between two sequences is a computationally intensive task. To illustrate this, let's take the following example. Suppose we have two sequences, *A* and *B*. For simplicity sake, let's assume the sequences are of the same length. If the task is to simply check if the two sequences have anything in common, this task would be trivial since we can stop immediately upon finding a single match. If additional parameters are added in the search criteria, such as minimum length, we essentially increase the complexity by that many folds. In the simple case, we can follow these simple steps: First, we fix the starting location in one of the sequences (for example sequence *A*), and then we compare each letter from the second sequence, i.e. sequence *B*. Given that we do not have any criteria in the length of the substring, we can simply stop the search once we have found a match. If no match is found and we've reached the end of the sequence B, we can simply move to the next position in sequence *A* to repeat the process. If we traverse the entire length of sequence A, then this process has a worse case complexity of O(N^2). However, on average we expect to have a lower complexity. This complexity increases when we add additional parameters as part of the search criteria. One example one may consider is to find whether there exists a substring with a minimum length *k*. While the overall complexity does not change, the number of comparisons that we have to perform drastically increases, consequently, increasing our constant associated with the complexity. This complexity increases even more if we are interested in finding all longest possible common substrings. Obviously performance is much more computationally demanding when we're interested in finding all of the substrings versus having to simply detect whether there exists a substring or not. Dynamic programming is frequently the preferred method when dealing with small dataset versus the naive approach as it performance fewer comparisons. However, one of the caveats of dynamic programming as mentioned earlier is memory consumption. Depending on the data representation, memory consumption can grow quite large. Needle man [33] *et al* and Smith-Waterman [24] are two algorithms that to this day are still widely used for sequence alignment.

2.2 Hardware optimization

2.2.1 Cache-aware vs Cache oblivious

A program is called a cache-aware algorithm when it contains explicit knowledge and hard coded parameters regarding the Central Processing Unit (CPU) hardware. A cache-aware algorithm is able to take advantage of software and hardware optimization as well. When data is requested to be loaded into the CPU's memory, *i.e.* cache memory, additional data is loaded besides what is needed by the CPU. This additional data, is considered future data that will be utilized at some point during the lifetime of the program. The reason for this additional data being loaded ahead of time is so we can minimize the number of times the CPU has to request (or fetch) data from the main memory. One of the main if not the main bottleneck of any software program is what's so called the input and output (I/O). This comes as a result in the significant difference in speed (latency) between the CPU cache memory and random access memory that exists in the hardware to date. A typical computer, i.e.. personal computer (PC) or server, will obtain its highest latency if the requested data does not exist in the main memory, rather it has to fetch it from the hard disk (HD). HD being the slowest of all the different types of memory, it is the cheapest and largest in size - spanning in the size of several terabytes. On the contrary, main memory is more expensive than HD but much faster. Unfortunately, main memory is significantly smaller, spanning only in the tens of gigabytes. A third type of memory is called cache memory, which resides inside the CPU. In contrast to HD and main memory, cache memory is by far the most expensive, however, it is the smallest and fastest. The size of cache

DNA encoding

Alphabet	A	C	G	T
binary value	00	01	10	11

Table 2.0: DNA Alphabet represented as binary values utilizing only 2-bits

memory, is extremely small. The overall size can be up to 30-40 (for level-3 cache) megabytes (MB). Cache memory is also divided into three levels: 1, 2 and 3. Levels 1 and 2 are only a few megabytes is size, however much faster than level 3 cache. Levels 1 and 2 are also much smaller than level 3 cache spanning only couple MB in size (to date we still see personal computers with levels 1 and 2 cache with sizes less than 1 MB in size.

2.2.2 Data movement

When the CPU needs to process data, first it looks into the cache memory. If the data is found, then data is processed immediately. Otherwise, it will check if the data exists in the main memory and hard disk only if data is not found in the main memory. Due to the significant latency difference between the cache memory and other memory types, minimizing the number of memory data fetches plays an essential role in high performance computing. Consequently, by having intimate knowledge of cache parameters, we can have maximal control of how much and what data we fetch into cache memory, giving us optimum performance - in theory. In theory, and perhaps practice, loading specific amount of data into cache may be sufficient when the data size itself is small, however if the data size becomes large enough where it does not fit into memory, then it is inevitable avoiding what is so called a cache-miss. Ideally we always want to have cache-hit, *i.e.* data is always found in cache and there is never a need to go to main or hard disk memory.

While cache-aware algorithms can benefit from both, software and hardware optimizations, when the data is not in the cache, having control as to how much to bring into cache may not be the ideal case. Since cache memory is so small, this space is shared among many other concurrent running processes at any given time. Thus, because we don't know what any of the other processes (other users, and system) are doing or their resource demands, our data can still be off loaded due to the resource scheduling by the operating system - consequently and potentially leading to a high rate of cache-miss (although we may have optimized the amount of data to load into the cache for our process). Thereby, having intimate knowledge of the cache parameters, not only will it not be computationally beneficial, but it increases the difficulty during the software development stage, limit portability and as well as maintenance of the application. With technology advancements that have been made in the recent decade (software and hardware), the amount, volume and dynamics of data generated is enormous compared to even one decade ago. It is noteworthy to mention that optimization advances at both levels, operating system, i.e. software, such as compilers, and hardware (faster chips and multi-cores) have made significant strides allowing scientists to tackle problems that were once considered computationally impossible. Thus, while software optimization is crucial, by having hard coded parameters, application becomes obsolete with respect to being limited to run only on the same type of hardware. Moreover, the application does not benefit from any other type of hardware and/or system resource scheduler optimization.

Cache-oblivious algorithms are completely oblivious of the CPU system hardware. One of the advantages that a cache-oblivious algorithm has over a cache-aware algorithm, it is independent of the hardware, thus it allows scientists to use their methods in almost any system or hardware. Another benefit is the implementation complexity is significantly less whereas when trying to incorporate cache size parameters as part of the optimization process. Consequently, the optimization responsibility is placed on the person who develops the software. When dealing with large enough data that cannot fit into cache memory or main memory, one of the properties that must be true is data linearization. Data linearization is a concept which shows how data that needs to be processed is accessed. Imagine if we have a list of numbers needing to be processed, and we access them from beginning to end. When the CPU needs to process for example the 34th element, and this number is not in cache, then the CPU has to fetch the data from main memory. However, due to hardware optimization, we not only fetch data needing to be processed, i.e. the 34th element, we fetch a chunk, i.e. spanning some distance; for example, we fetch elements 34 to 40. Given that we only process each element once and always move in the forward direction, we essentially minimize the amount of cache-miss that the CPU encounters when data is larger than cache can store. Thus, are able to achieve the same type of optimization as cache-aware without having explicit knowledge of the hardware. An example where data is processed not in a linear fashion is when we process our elements in random order. Same example as above, however, now instead of processing each element in order, we randomly choose an element; upon processing it, we randomly chose another element, until we have processed all elements. If we need to process a large list of numbers, because we do access each element randomly, we can no longer guarantee cache-hit. This phenomena occurs because when we fetch the 34th element from main memory, we end up fetching elements 34 to 40. However, due to the randomness, if our next element that needs to be processed is elements 15, CPU would suffer from cache-miss since it would have to go back to main memory and fetch the element. It is easy to see how memory is not accessed in a linear fashion. While this approach does not take advantage of the hardware

SEQ	Binary	Decimal	SEQ	Binary	Decimal	SEQ	Binary	Decimal
A	00	0	$(C)_3$	10101	21	AATGCT	000011100111	231
C	01	1	$(A)_4$	00000000	0	AGTGTCA	00101110110100	2,996
$(A)_2$	0000	0	$(C)_4$	1010101	85	ATGGGGT	00111010101011	3,755
$(C)_2$	0101	5	$(A)_5$	0000000000	0	GTAGATAA	10110010001100	45,616
$(A)_3$	000000	0	$(C)_5$	0101010101	341	TTTTTTTT	11111111111111	65,535

Table 2.1: Conversions of DNA sequences to decimal number system

optimization, both approaches, i.e. cache-oblivious and non cache-oblivious, can run on almost any system.

2. 3 Naive approach method protocol

With our Cache-Oblivious in Memory Computing for Elements of Extreme Conservation algorithm, we are able to find extremely conserved elements not only in chromosome assembled genomes but in chromosome unassembled genomes as well. For chromosome assembled genomes, input must be a directory containing all of the FASTA file format [27] (chromosomes files), where for chromosome unassembled genomes, only the file that contains the genomic sequence fragments (contigs and/or scaffolds) is necessary. For the naive approach, the complexity to find all substrings common between two sequences is $O(N^2)$. Problem definition: Given two sequences, find all unknown substrings common to both sequences:

We start by comparing some character s_1^i with character s_2^j. If both characters are the same, then we compare the second character in S_1 with the second character in S_2, s_1^i+1 and s_2^j+1. If characters in both sequences are the same, we repeat the process until the characters no longer match. At this point, if the length of the substring is greater than 0, we have found a substring between S_1 and S_2. Upon finding a substring, we increment s_1^i to s_1^i+1 and reset s_2^j, where j = 0. We repeat this process until we reach the end of S_1, where s_1^i = n. However, if the

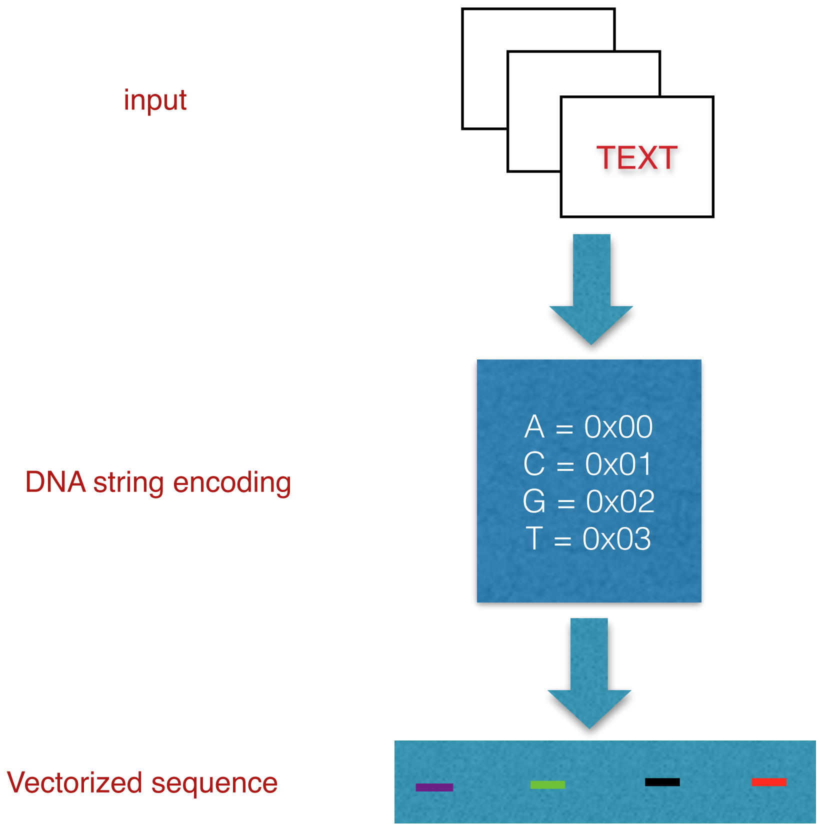

Figure 2.0: Vectorized sequence protocol. String sequence is encoded as a number representation by taking each letter and converting it into a number. We perform the conversion in chunks at time by selecting 12 characters at a time.

character in s_1^i does not match the character in s_2^j and the number of characters matched thus far is zero, then we skip the current character in s_2^j, and compare characters s_1^i and s_2^{j+1}. If the two characters still do not match, increment s_2^j again, and repeat the process until end of the sequence S_2, where $s_2^j = m$. Once we reach the end of sequence S_2, we increment index of sequence S_1, s_1^{i+1}, reset s_2^j, where $j = 0$ and repeat the process. On the contrary, if the two characters match, we increment both indices, *i* and *j* by 1, to compare the next two characters. We repeat this process until we find the first mismatch, which at this point if the length of matched strings is greater than zero, a substring is found. We repeat this process until we have reached the end of sequence S_1, thus guaranteeing to have found all substrings that exist between sequences S_1 and S_2. It is noteworthy to say that in practice, when finding substrings, we typically have additional constraints, such as finding all substrings with some minimum length.

One of the benefits of the naive approach, it is very simple to implement. Since we make use of the original data, the amount of resources needed does not increase. However, while this approach suffices for small strings, it becomes impractical when sequences become very large, such as genomic size sequences. Furthermore, one of the biggest disadvantages of the naive approach is the fact that we compare every single character in order to determine whether or not some substring (for some minimum length) exists between two sequences.

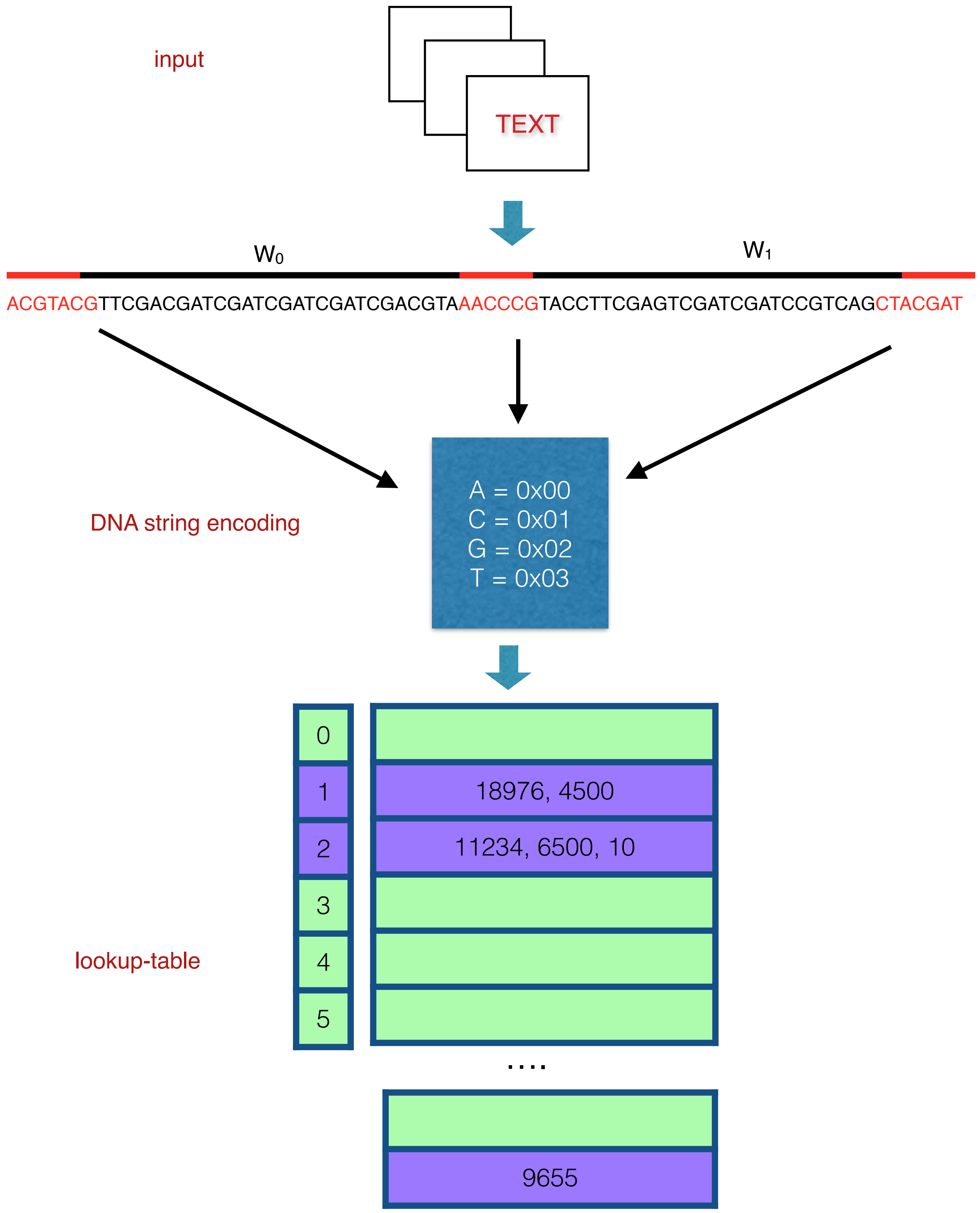

Figure 2.1: Hash-map lookup-table sequence protocol. String sequence is encoded as a number representation. First search-key (in red) is used as an index into the hash-map lookup table to access the correct bin, whereas the second search-key is what is stored into the table

2.4 Hybrid approach protocol

2.4.1 Obtaining sequence data

Sequence genomes, chromosome assembled and unassembled, were downloaded from the Ensemble database (Flicek *et al* 2013) [26] and from the National Center for Biotechnology Information (NCBI) (Wheeler *et al* 2005) [27]. From our dataset, all but one of the species were from the Ensemble database build-73, whereas Elephant Shark was downloaded from NCBI build-6.1.3. All genomes were downloaded in the fasta file format (Lipman *et al* 1985) [25]. When available, gene sets for each species annotations of both coding and non-coding genes were downloaded. Assembled genomes contained one chromosome per file whereas for unassembled genomes a single file contained all contigs and/or scaffolds.

2.4.2 Data Preprocessing

Sequences from the entire dataset are converted (represented) as numerical values. Starting from nucleotide at position 0, we select a substring of length *k*, where *k* is the number of base pairs that will be converted into a numerical value. When dealing with small number and/or size sequences, string comparison can be achieved in a feasible amount of time. However, for

a) ACGTTAGTACCCAAGTTGACCCATGCATGCGA

$\downarrow$

b) AGCGTACGTACCCAGTTGAACCCATGATTGCA

$\downarrow$

c) TCGCATGCATGGGTCAACTTGGGTACTAACGT

Figure 2.2: Protocol for reverse complement. (a) original input sequence, (b) reverse of sequence in ‘a’, and (c), complimentary of sequence ‘b’.

large genomic type sequences, our performance decreases significantly, thus making LIME detection impractical if not impossible. Thus, to increase performance, one of our optimizations we employ is converting the genomic data from character representation into numerical values during the comparison. Moreover, not only numerical values are more efficient when comparing large sets of elements, they can also be utilized as indices and reduce the amount resources. Given our small number alphabet, *i.e.* only four letters, we are able to represent our nucleotides (A, C, G, T) using only two bits whereas a single character occupies 8-bits, **TABLE 1.0**. Our conversion function in which performs the conversion of each character in the sequence, represents each nucleotide as the following:

Given a sequence S, where $S = \{s_0 s_1 s_2 \ldots s_{|S|-1}\}$,

$$f(S) = \sum h(s_{i,} b_i), \text{ then} \tag{2}$$

where S is the nucleotide sequence, *s* is the individual nucleotide, *h* is the bit manipulation conversion function and *b* is the bit value. One of the advantages utilizing bit representation is because we can achieve a one-to-one mapping between string representation and numerical values, with the exception of a single letter ‘A’ and the sequence of letters ‘AAAAA…’, however, this does not lead to any issues **TABLE 1.1**

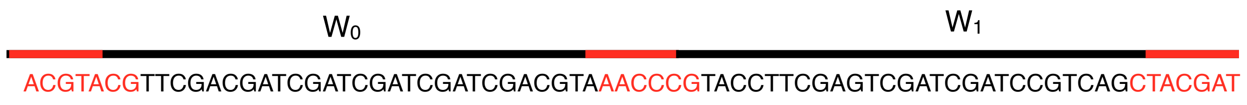
W₀
W₁
ACGTACGTTCGACGATCGATCGATCGATCGACGTAAACCCGTACCTTCGAGTCGATCGATCCGTCAGCTACGAT

Figure 2.3: Amount of information stored in the vectorized sequence. Only the search-key (in red) numerical representation is stored. The distance between search-keys is determined by the formula in equation (2).

2.5 Data structure optimization

To date, the definition of Big Data is still quite not completely defined. Many agree however, that for something to be considered Big Data, it will most likely be computationally demanding and/or infeasible with respect to time. Less than a decade ago, most domains were not involved in Big Data. However, with the technology ever improving, as recent as in the last couple of years, the rate of which the amount, complexity and the diversity of the data that are generated have made many biological (and many other domains) problems computationally very expensive if not infeasible (even with faster processes, and parallel computing); thus we can no longer approach the same problems with a one size fits all type solution.

Such increase in data generation is worthy of attention particularly in the field of Genetics. Not too long ago was when we were able to sequence genomes at resolution of 5-fold the genome size [2], however, now we are able to sequence genomes at unprecedented depths, thus generating an enormous amount of data. As a consequence, not only should we optimize at the software level, we must also optimize at the hardware level in order to support current and future demands. The third parameter that is perhaps the most overlooked, is data optimization.

Due to the increase of computational resources, in the field of Computer Science the emphasis has been put more on software rather than data optimization. Particularly in the era of Big Data, data optimization plays significant role in the performance of an algorithm, thus it is pivotal to not only choose only the necessary needed data, but how we represent our data as well. For example, when comparing two strings, then number of CPU cycles is significantly higher versus numeric comparison.

In genomic data, often times we can take advantage of the small alphabet side. By representing our DNA string sequences as numerical values, not only we achieve better comparison performance, we are also able to reduce the amount of storage required for each of

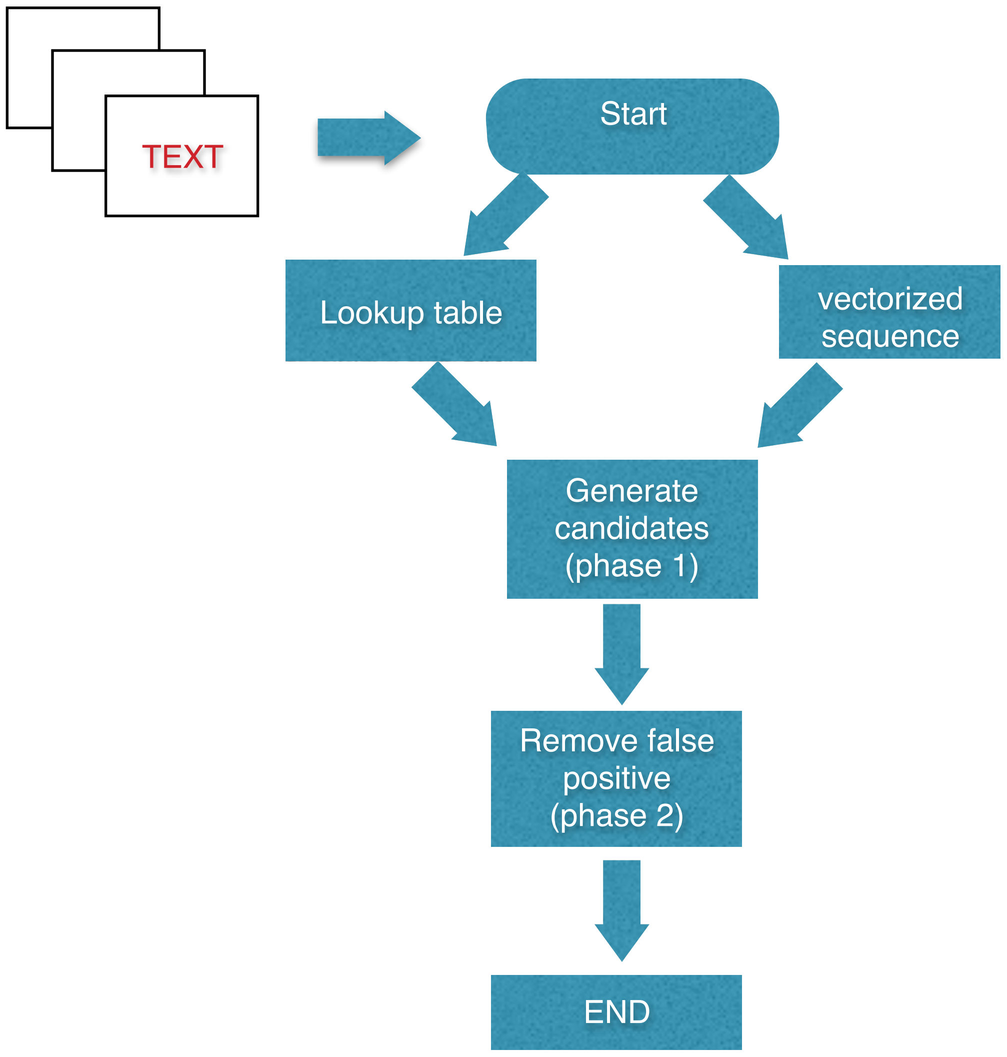
TEXT
Start
Lookup table
vectorized sequence
Generate candidates (phase 1)
Remove false positive (phase 2)
END

Figure 2.4: Overall cache-oblivious pipeline. Given sequences (a) and (b) are our input, once raw DNA sequence is loaded in memory, we construct the vectorized data structure using sequence (a), and hash-map lookup-table using the sequence (b) per protocol. Phase I is generated by matching the prefix and suffix between substrings from (a) and (b) respectively. Phase II explores the intermediate data between the prefix and suffixes and removes any discards any substrings that are less than the minimum threshold.

the chromosome sequences. Employing numerical representation, we only need two bits where string representation requires eight bits for each letter. Thus, we can improve the amount of storage needed for the same information by 4-fold data reduction - this is our first step of our data reduction.

2.5.1 Vectorized sequence representation

Our second optimization steps that we employ is by utilizing streamlined and indexed data structures, utilizing the numerical representation as our indices, i.e alleviating usage of an external hash function (no need to utilize a hashing function since our numerical values are used as hashes themselves).

When comparing two sequences, where one sequence belongs to species x, and the other sequence belongs to some species y, each respective sequence is represented as a two different data structures, i.e. species x data representation will be different than species y. Thus, given two sequences, ‘a’ and ‘b’, we represent sequence ‘a’ as a vectorized numerical sequence and sequence ‘b’ as a hash-map lookup table - described below. Given a sequence in it’s raw string form, a vectorized numerical sequence is generated by converting 12-bp at a time as numbers (search keys), see Eq. 2. Given a sequence with n-bases, we then would be able to generate n-k-1 search keys (**Figure 2.0**) in which are used to query our hash-map lookup table.

2.5.2 Hash-Map lookup table

Our second data structure representation is a hash-map lookup-table utilized for the second sequence, ’b’. Given that our hash-map lookup table is a two-dimensional structure vs linear (vectorized sequence), we keep the number of bins finite, i.e. the range in which our search keys represent (**Figure 2.1**). For example, if our search key is generated from 5-bp, then the number of bins in our hash-table is 1,024, and for a search key generated from 8-bp, we then generate 65,536 bins for our hash-map lookup table…etc, thus with each increase in basepair, we increase the number of bins by 4-folds, which is to no surprise given that our alphabet size is 4. Given n-bp, then total number of bins is equal to 4^n.

Similarly as in the vectorized sequence representation, we generate n-k-1 search keys. For example, at nucleotide position 0, we generate our search key by extracting a word with length k, from position 0. Provided minimum length for a LIME (given by the user), we also extract and generate a search key using the last k-letter word. Thus if the minimum length is 100, and the search key is generated using 8-bp, then the two search keys would be from the ranges 0-8 and 92-100, where the beginning (or prefix) ranges from 0-8 and the end (suffix) ranges from 92-100. Our prefix (extracted from position 0-8), then is encoded as a numerical value that is used as an index into the hash-map lookup-table to access the particular bin containing a list of tuples (these tuples have the same exact prefix). Each tuple in our list stores two values, a) the index in which the prefix is located (in this example 0), and the suffix **Figure 2.1**. Ambiguous bases (bases other than A, C, G or T) are allowed in sequences in the FASTA file format. For instance, R = (A or G), Y = (T or C), and K = (G or T), etc. However, because we cannot assume that a particular character is one way or another, we ignore any ambitious nucleotide characters. Thus, if our search key (prefix) contains any ambiguous letters at any position, we simply discard it.

2.6 Reverse-compliment protocol

DNA sequence can be read in the forward and reverse direction thus one can obtain the same sequence from either direction. Complimentary of two strands shows the relationship of two structures using the key-lock principal. Complimentary base pairing allows a cell to replicate itself or repair damage to the information stored in the sequence, thus it is necessary to find LIMEs not only in the forward direction, but in the reverse-complement as well, **Figure 2.2**. To generate the reverse-compliment, first, we reverse the sequence, i.e. the last character becomes the first, second to last character becomes the second and so on. In addition, each character then is replaced with its complimentary form, such as A <=> T and C <=> G.

2.7 Information retrieval using cache oblivious approach

In the following subsection, we describe in detail each step of the cache-oblivious algorithm design. Each species is preprocessed according to the protocol in the data structure section. Our algorithm does a pair-wise comparison between two assembled, un-assembled, or mixed between assembled and un-assembled genomes. We designate one of the species as the query genome, whereas the second genome is designated as the database genome in which will be queried by the query genome. The hash values in the vectorized sequence are used as indices into the lookup table. Because our database contains all of the raw sequence information (in numerical form), the vectorized sequence stores only about 2% of the data by extracting and storing the search keys based on the following property:

$$W = L/2 - w/2, \text{ where} \tag{3}$$

L = minimum length,
w = search-key word-size
W = # of bases being skipped

Using equation (3), we guarantee that a subsequence common between two sequences will be detected. Such guarantee comes from the fact that there are at least two search keys (prefix, and suffix) spanning the common subsequence **Figure 2.3**. As a consequence we are able to ignore all of the data in between the search keys. Thus, our vectorized sequence generates the search keys by skipping every $|W|$ number of bases, where each window contains two search key, prefix and suffix.

First we load our raw data sequences into memory. Once both sequences are loaded into the system, we perform the data transformation, i.e. convert sequence *A* and sequence *B* into a vectorized representation and a hash-map lookup-table respectively. Using a search-key word of length 8-bp, our hash-map lookup-table will contain exactly 4^8 bins. Given that the lookup-table contains exactly the number of bins as the range in which the search keys are generated, i.e. 4^8, we utilize the search keys directly as indices to access the appropriate bin.

Our algorithm traverses the entire vectorized sequence in a linear fashion up to N-1, where N is the number of search keys. Each element *i* in the vectorized sequence is used as the index into the lookup table to retrieve the list of tuples. Furthermore, if the tuple list at the index bin is empty, we are guaranteed that no LIME exists with the prefix as the search key. By comparing each of the tuple's search key with the suffix of the vectorized query sequence (with element *i+1*), we essentially compare the beginning and end points of the substring between the two sequences. This process is called coarse grain filtering as it alleviates the need to investigate the entire window. By matching only the end points we drastically reduce the number of candidates that we explore further during the fine grain filtering phase.

Our second stage of LIME generation is to investigate all of the LIME candidates in which we found during the coarse grain phase by exploring the intermediate elements of the subsequence in question between the two sequences. If the number of matched characters is greater than the minimum LIME length, we continue expanding in both directions, to the left and right of the sequence, until the first mismatch is found. However, if the number of matched characters is less than the minimum number of characters, our algorithm simply discards it; otherwise, we extract the exact position in which the common subsequence exists in the respective parent sequence. It is important to note that given that DNA is a double helix, we must search for LIMEs in the forward and reverse-complement directions. From a biological point of view, it is important to know the strand direction (in which the LIMEs are generated), as the genetic structure in the forward direction vs reverse-complement direction are different. Each strand holds different biological properties and functions, thus, this process is repeated twice as it is necessary to also scan the query sequence in the reverse-complement order **Figure 2.2**. The generated output contains the LIME(s) position in the sequence and length, **Figure 2.4** shows a pipeline of the entire process.

2.8 Method pseudocode

1. Generate vectorized numerical sequence from sequence A.
2. Generate numerical hash-map lookup table from sequence B.
3. Generate potential substring candidates.
4. Filter any false positive candidates, thus leaving us with all longest possible substrings that are common between sequences A and B.

```
DETECT_LIMES(DIR1, DIR2) {
        species1_file_list =: get_files_from_dir(DIR1)
        species2_file_list =: get_files_from_dir(DIR1)
        FOR i =: 0 to N DO
            FOR j =: 0 to M DO
                LIME_GENERATION(species1_file_list$^{i}$, species2_file_list$^{j}$)
            END FOR
        END FOR
}
```

```
LIME_GENERATION(S1, S2) {
        s1_vector = convert_sequence_to_scalar(S1)
        lookup_table = generate_lookup_table(S2)
        candidates = generate_candidates(s1_vector, lookup_table)
        candidates = filter_candidates(candidates)
        remove_duplicate_positions(candidates)
}
```

Step 1 pseudo code - generate scalar sequence:

```
convert_sequence_to_scalar(sequence) {
        v =: ()        //empty vector
        FOR i =:0 to N DO:
            s = getSubstring(sequence, i, 8)
            hash =: generateHash(s)
            addHashToVector(v, hash)
            i+=: |W|+|w|
        END FOR
        return v
}
```

Step 2 pseudo code - generate hash-map lookup table

```
generate_lookup_table(sequence) {
    lookupTable = ()     //empty vector of vectors
    FOR i =:0 to $4^8$ DO:
        addVectorToLookupTable(lookupTable , v())
        i+=:1
    END FOR

    FOR i =: 0 to N-|w|
        prefix =: getSubstring(sequence, i,8)
        suffix =: getSubstring(sequence, i+|W|, 8)

        v1 =: validate(prefix)
        v2 =: validate(suffix)
        IF v1 AND v2 THEN
            preffixHash =: generateHash(prefix)
            preffixHash =: generateHash(suffix)

            lookuptable[hash].addSuffxWIthIndex(suffix, i)
        END IF
        i+=:1
    END FOR
}
```

Step 3 pseudo code - generate potential substring candidates:

```
generate_candidates(hash_vector, lookuptable)
    candidates =: ()        //empty vector
    FOR i =: 0  to  N-1 DO:
        suffixA =: v[i+1]
        subset =: lookuptable[v[i]]
        FOR j =: 0  to  M DO:
            suffixB = subset[j].suffix
            IF suffixA = suffixB THEN
                addCandidate(candidates, i*|W|, suffixB.index)
            END IF
        END FOR
    END FOR

    return candidates
}
```

Step 4 pseudo code - remove false positive substring candidates:

```
filter_candidates(candidates) {
  candidates =: ()         //empty vector
  FOR i =: 0  to  N DO:
      indexA =: candicates[i].indexA
      indexB =: candicates[i].indexB
      FOR j =: 0 to M DO:
          IF A[indexA] = B[indexB] THEN
              numberOfMatches +=: 1
          OTHERWISE
              EXIT LOOP
          END IF
          j +=: 1
      END FOR

      FOR j =: M to 0 DO:
          IF A[indexA] = B[indexB] THEN
              numberOfMatches +=: 1
          OTHERWISE
              EXIT LOOP
          END IF
          j -=: 1
```

```
            END FOR

            IF numberOfMatches < THRESHOLD THEN
                removeCandidate(candidates, i)
            END IF
            i +=: 1
    END FOR
    return candidates
}
```

Reneker et al 2012 dataset

	Human	Mouse	Rat	Chicken	Dog	Macaque
# of Chromosomes	24	21	21	31	39	21
Size	~3GB	~2.6GB	~2.8GB	~1GB	~2.2GB	~2.7GB

TABLE 3.1: Mammalian data set

Method comparison using same hardware

	Hybrid	Reneker *et al* 2012
Lewis cluster	~16 days	21 to 28 days
PC Desktop	~30 hrs	n/a
Cores	1	48

Table 3.2: Method comparison using same exact computer hardware architecture. Based on our above results, we show how much of a role the hardware optimization plays in computational power, thus hiding the true difference in performance between two algorithms

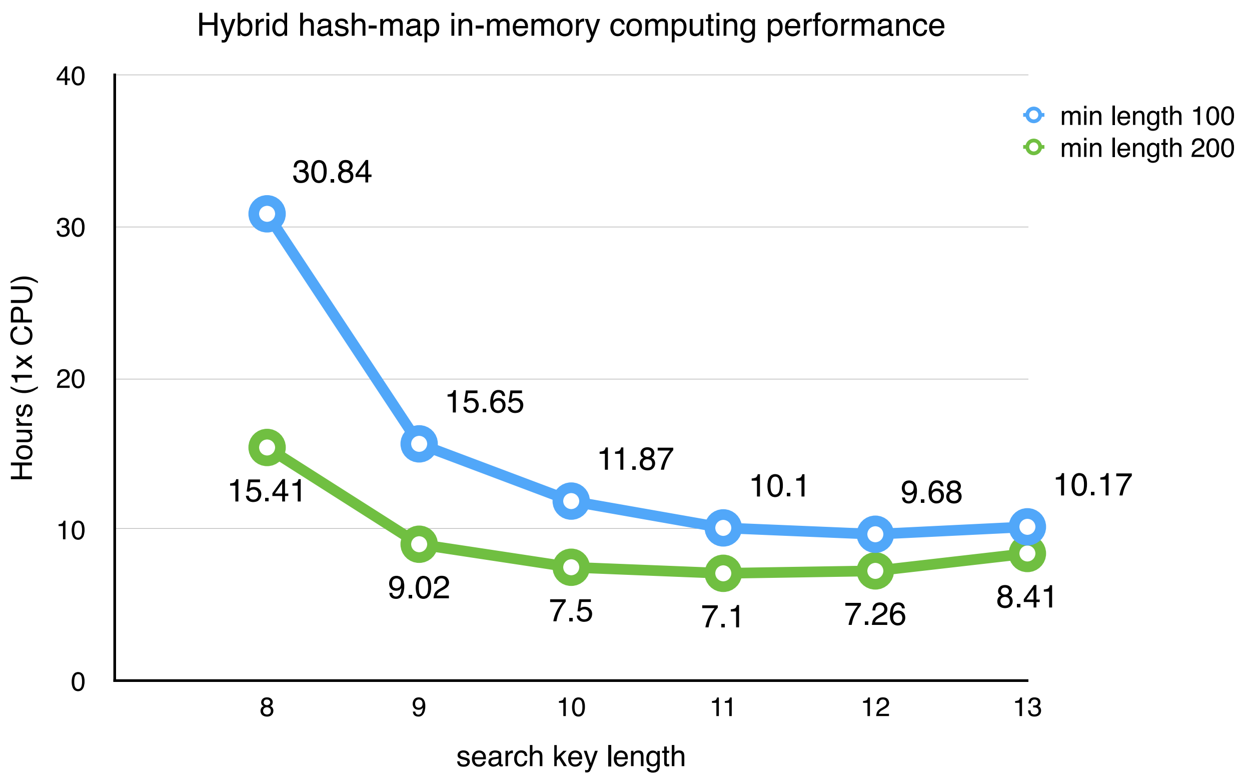

Figure 3.1: Overall performance using search-key length between 8 to 13 characters.

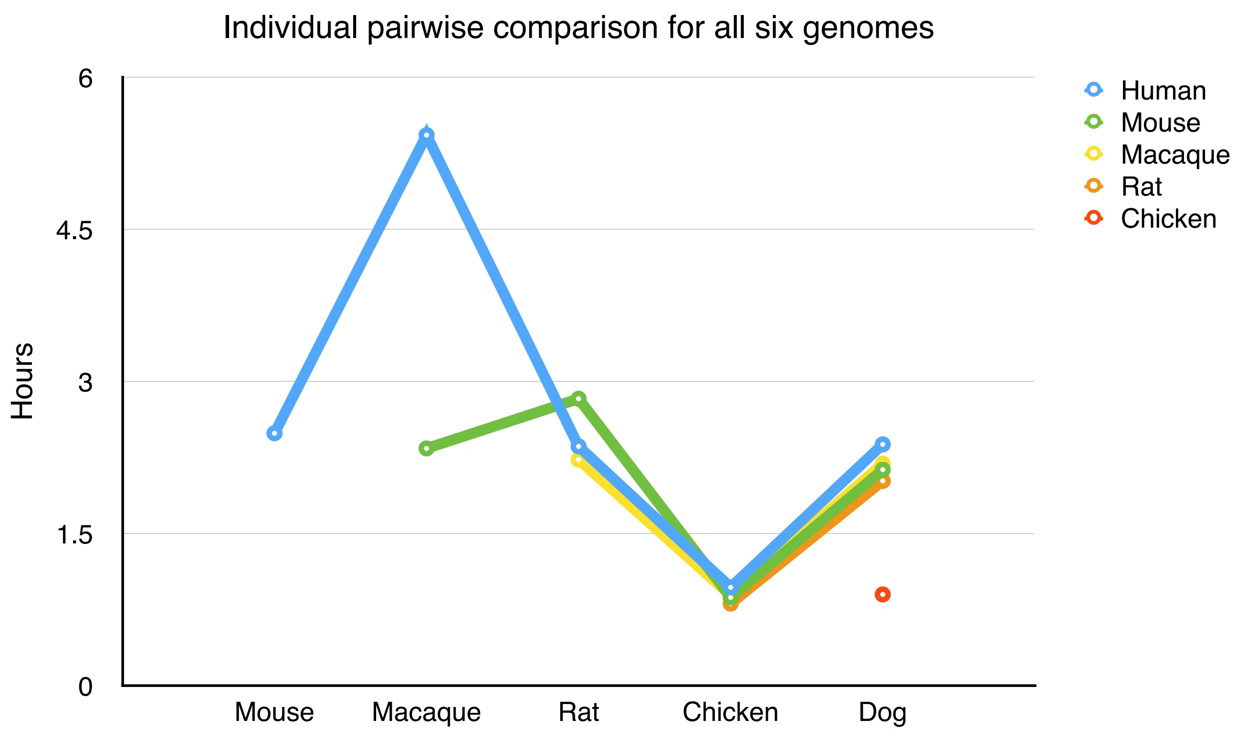

Figure 3.2: Individual pairwise comparison for all size genomes all against all, displaying algorithm behavior between any two species

CHAPTER 3

Assessment

3.1 Hybrid approach

The search key size, i.e. prefix and suffix length play significant role in the performance of our new hybrid algorithm. During our initial phase of the coarse grain filtering, prefix and suffix significantly reduce our search space without having to visit all characters in the sequence. In addition, number of bins in our hash-map lookup-table is always constant, despite of the genome size, predetermined by the search key size. The more information we pack (represent) as numeric values, the more we reduce the total size of our data in-memory representation. Thus, if we represent 16-bp as a single integer (2-bits per character versus 8-bits), i.e. occupying the full 32-bits, we reduce our search space significantly more whereas in the case where we'd only utilize 8-bp. While search space may decrease significantly, however (since we cover a much larger part of the sequence via our prefix), at the same time, we increase the amount of resources needed to generate the hash-map lookup-table. Previously, we mentioned that one of the benefits for utilizing the prefix as an index in the lookup-table is to alleviate the need to have a mapping function between numeric representation and indices for the hashes generated from the original sequence. Consequently, if we increase the word size to 16-bp, we increase the length of the lookup-table by 4^8-folds (~65K), a much larger structure than when using a word-size of only 8-bp. Another disadvantage of having a very large hash-map look-up table is the sparsity of the data, thus memory resources get allocated when not in need or that the benefits of having such larger structure get absorbed by the time-complexity and the amount of resources it takes to generate the structure. For this reason, we have to obtain a balance by optimizing the performance via the amount of data reduction we gain by increasing the word size; at the same time as we minimize the increase in complexity that is caused due to increase

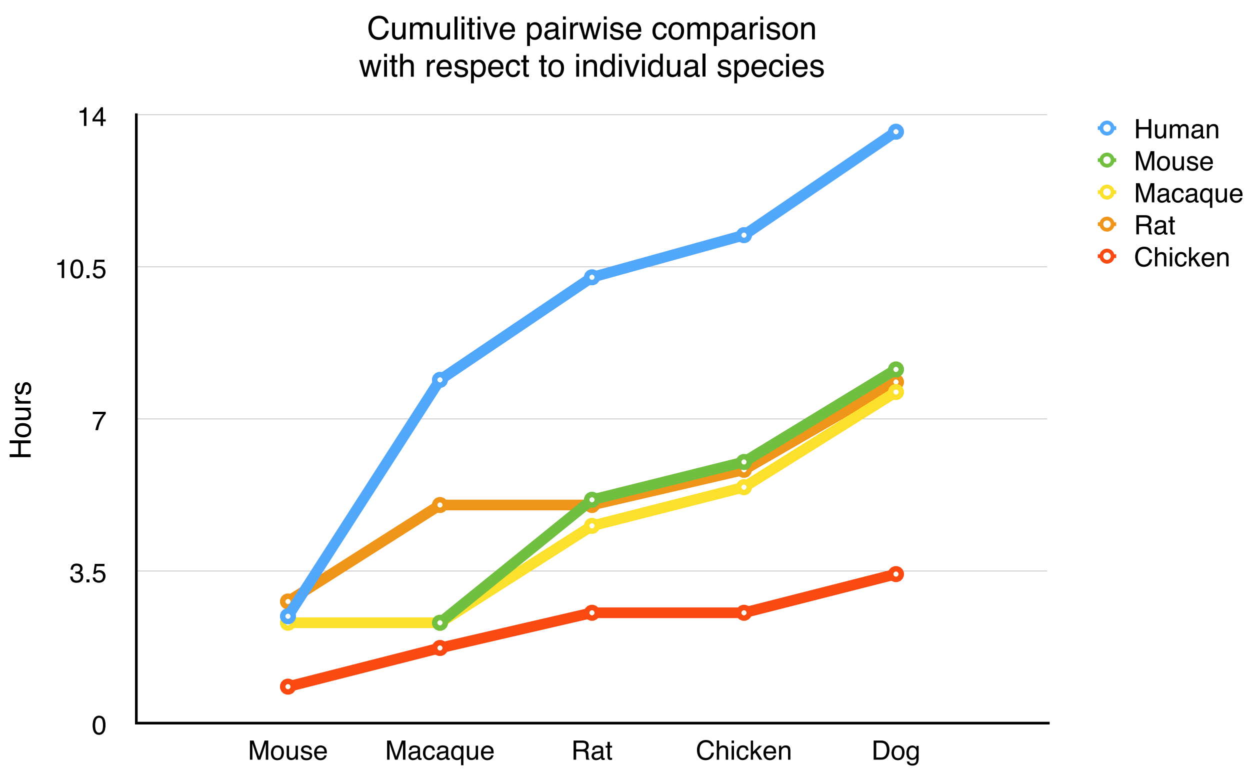

Figure 3.3: Time comparison for a species with respect to time when one addition species is added to the comparison

in word-size. We evaluated our method by selecting word-sizes in the range between 8 and 15 characters long. As part our evaluation process, use used the same species as were used in Reneker *et al* 2012. Reneker *et al* 2012 analysis included two different groups of genomes that were used as part of their data analysis, i.e. animals and plants genomes. Each group contained six species, all of which were assembled. The animal group constituted of the following species: *Homo Sapiens* (human), *Mus Musculus* (mouse), *Rattus Norvegicus* (rat), *Gallus gallus* (chicken), *Canis Familiaris* (Dog), and *Macaca Mulata* (Macaque). For the plant group, the following species were utilized: *Arabidopsis* (Arabidopsis thaliana), *Soybean* (Glycine max), *Rice* (Oryza sativa), *Cottonwood* (Populus trichocarpa), *Sorghum* (Sorghum bicolor), and *Grape* (Vitis vinifera) **(Table 3.0).**

	Human	Rat	Coelacanth	Puffer fish	E. Shark	Lamprey
# of Chromosomes	24	21	21	31	39	21
Size	~3GB	~2.8GB	~2.9GB	~233MB	~850MB	~900MB

TABLE 4.1: Tetrapods, jawed and jawless fish data set. Species highlighted in red indicate chromosome unassembled genomes

Using an all against all type comparison, the alignment-free approach from Reneker *et al* 2012 was able to obtain the results for the animal group in approximately 3 to 4 weeks of processing time, while for the plants group, it took approximately one week using the lewis cluster [27] hardware. The word size as part of it's hash-indexing component was set to 8-bps, as it achieved best performance. Since we were provided with the source code [15], it gives us the ability to compare not only the methodologies between our-hybrid and the original hash-map approach [15] from the end-result point of view, but we are able to run the exact implementation code on the same computer architecture (hardware). This is extremely important as often times, different implementations (although following the same methodology) often give different performance results (alleviating hardware improvement). As part of their preprocessing step, the original method generates two types of file output: an index file for the hash-map. Hash-map file contains all of the preprocessed chromosome information - one index and hash-map file per chromosome. It is important to note that while the index file for the hash-map is small, the database hash-map file itself for species such as human, mouse…etc, the total size for the entire genome is approximately 50GB for each genome, thus for each pair comparison the algorithm has to utilize ~100Gb of data from the hard drive. This is rather inefficient as we load significant amount of data each time we compare one chromosome from one species to another chromosome of another species. Given that the hash-map database-like files are large, each time we load a hash-map for a different chromosome, it has a high likelihood of causing a cache-miss as only a small amount of information can be stored at a time, causing the CPU to be occupied more with the data movement than the actual processing of the data. As a consequence, even for smaller size genomes, performance gets significantly hindered. In our hybrid approach however, we are able to reduce our input dataset on average by half due the prefix generation, i.e. skipping every |W| base pairs. Our vectorized data structures stores only 2% of the sequence information whereas the lookup table retains 100%, allowing us to reduce the overall data usage to approximately 50%.

3.2 Data reduction

By representing our data as numerical values we are able to reduce our memory footprint by 4-fold, in theory, however, in practice typically we reduce our data by 2-fold. Since the DNA alphabet is only 4-letters, we are able to represent each nucleotide letter as a numerical value, using only 2-bits **(Table 1.0).** Thus, since each character is represented as 8-bits, we are able to store 4-times more data as a numerical representation. However, depending on the datatype representation, i.e. *short or int* (occupying 16 and 32 bits each respectively), we then would suffer from allocating memory that does not get entirely used. For example, in 16-bits, we are able to store 8 DNA nucleotide characters which are equivalent to 64-bits, thus we achieve a 4-fold reduction. However, if we were to use an integer datatype, where it occupies 32-bits, then for the same 8 DNA nucleotide characters, we only gain 2-fold in data reduction. Moreover, if we use a data structure of type long (8-bytes, 64-bits), we would not gain in memory usage at all. To obtain maximum data reduction, one has to fully understand the advantages and limitations of employing the different datatypes as it is imperative for high performance computing.

3.3 Hybrid approach vs current state-of-the-art

To have comparable benchmarks between the two methodologies we ran our method on the Lewis cluster [29], the same architecture as Reneker *et al* 2012. Additionally, we utilized the same search key word-size of 8-bp - following the same standard as the original approach. For the animal group species we were able to obtain the same results in 31hrs hours. resulting in 31-time speed, utilizing only a single processor, whereas utilizing the same number of process, i.e. 48, we would be obtain our results in approximately 47 minutes, 643-fold speedup. In order to have an accurate benchmark between two different methods, it is imperative to measure our method performance employing the same or similar system hardware since the computer architecture plays a significant role in performance. Thus, we measured the difference in performance between the personal computer desktop and the Lewis cluster [29]. We were able to process all six species from the animal group in 16 days (396hrs) - resulting our hybrid approach to perform approx. 12-times slower than the personal computer desktop, showing the significance that computer hardware plays in computing power.

Even though the performance of our hybrid approach showed only ~50% improvement, we utilized only a single core, wheres the original approach processed the same data sets in approximately three to four weeks utilizing 48 processes simultaneously. In the event we utilize the same number of processes on our hybrid approach, we would be able to process the same dataset in approximately 8 hrs; resulting in approx. 63-fold speedup. Being able to achieve this type of speedup allows us to significantly expand the number of genomes that we can process, opening the door to a new evolutionary frontier in the scientific community. It is important to note that the number of assembled genomes is significantly less than the number of unassembled (but sequenced) genomes, becoming problematic with respect to computation scalability. This comes as a result of the number of searches we must perform. For example, if we have to process 10,000 tasks and each task takes 0.05 seconds, then the overall time is approximately 8 minutes whereas for 600,000 tasks, the time it would take to complete all tasks would be on the order of approximately 8 hours. This is significant since we perform an all against all type comparisons. Processing Homo Sapiens vs Elephant Shark, it would take in the order of 200 hours (roughly 8 days) to complete the process due to the large number of contig sequences in the Elephant Shark. It is not difficult to see now how rapidly the computation time increases, even when we perform one assembled vs one unassembled genome. For two unassembled genomes performance would be significantly worse. Thus, comparing unassembled genomes utilizing Reneker *et al* 2012 methodology,it would be impossible even for small number of genomes. To gain a sense of scalability of our new hybrid approach, we concluded a similar genome comparison as we did with the assembled genomes, however, in this case we utilized one assembled and one unassembled genome.

The species that were used for the unassembled vs assembled genome comparison benchmarks were Rat and the Coelancath fish using again a single processor. The Rat genome contains 24 chromosomes whereas the Coelacanth fish contains contains 22,819 contig sequences (due to no chromosome assembly to date). It took approximately 6hrs hours to process Rat against the Coelacanth fish genome utilizing our new hybrid approach whereas it took 6-8 weeks for the same dataset using Reneker *et al* 2012 approach, significantly much longer than any assembled pair species in the original method dataset. Consequently, it is not difficult to see that even processing only two unassembled genomes, it would be unfeasible if not impractical at all. As part of our benchmark and scalability **(Figure 3.3 and Figure 3.4)**, we utilized different word-size for our search-key, ranging from 8-13 bp. Unlike the current method where it uses 8-bp as part of its search-key, utilizing 8-bp our method achieved the worse performance - obtaining the results in approximately 31hrs, whereas fastest time was achieved utilizing a word-size of 12-bp, in just under 10 hours, i.e. a third of the time of the worse performance **(Figure 3.2).**

Chapter 4

Results

4.1 Performance

Using our regular personal computer desktop architecture with 8 gigabytes of memory and an i7 quad core processor, we achieved best performance employing word-size hash of 12 base pairs. On the contrary, our hybrid method achieved its worse performance when utilizing a word-size hash of 8-base pairs. Here in our work we have shown that even when utilizing a word-size of 8 base pairs, the performance is significantly faster than the current-state-of-the-art method. We were able to recall the full comprehensive set of LIMES, significantly better in both architectures, personal desktop and same architecture that the original algorithm was executed. Moreover, our hybrid approach utilized a single processor vs. 48 processors employed in Reneker *at al* 2012. If we utilized the same number of processors simultaneously, we would then obtain the same results in approximately 9hrs, resulting in an overall 63-fold speedup time. Our method performs even faster when using a word-size search-key of 12-bp with minimum length of 100-bp, reducing the time by two thirds. Having such improvement in performance, we allow the scientific community to conduct a much deeper and enriched analysis by covering a much larger number of species.

Using the original method we would be limited to analyze chromosome assembled genomes only, greatly reducing the amount of analysis between species. Furthermore, even if we utilized much faster hardware, the computation time exponentially increases as we add to our comparison even one single species. During our analysis between Rat and Coelacanth fish, our method performed 2016-time faster than the original approach [16]. Utilizing 12-processes on the Lewis cluster [29], the original approach took approximately six to eight weeks to complete the analysis between Rat and Coelacanth. We then compared the same pair species

Complex LIMEs copy number

LIME ID	Human	Rat	Puffer Fish	Coelacanth	E. Shark	Lamprey
206046	**11**	11	1	3	2	15
108000	11	11	2	3	28	15
205890	8	5	4	3	7	10
205926	8	7	5	3	3	12
207075	7	9	**9**	3	10	**16**
194342	7	9	9	3	8	3
164921	6	8	1	3	7	15
1376470	6	7	9	3	10	1
1368905	6	6	1	2	8	1
100233	5	4	3	3	2	7
106206	5	8	6	2	3	6
106315	5	4	3	3	20	1
1375914	5	6	4	4	17	1
97737	5	4	3	2	2	8
1368928	5	6	6	1	12	2
83286	5	7	8	1	12	13
205291	5	5	5	3	1	1
205976	4	7	1	2	2	15
206088	4	1	1	1	1	1
205793	4	5	9	1	1	3
1364042	4	7	6	1	14	1
205984	3	3	4	1	2	13
165901	3	8	1	1	2	1
165993	2	3	5	1	13	2
206717	2	3	6	1	1	15
1365033	2	2	9	1	8	10
106205	1	8	2	2	3	3
109312	1	8	6	2	3	6

Complex LIMEs frequency-continued

LIME ID	Human	Rat	Puffer Fish	Coelacanth	E. Shark	Lamprey
517363	**15**	1	1	3	65	2
204277	15	**86**	1	3	65	2
205180	8	5	4	3	2	10
205170	8	7	5	3	1	12
205169	8	7	5	**4**	2	13
58622	8	7	5	4	**103**	13
204692	8	5	4	3	2	10
37470	8	7	5	3	93	12
204536	8	7	3	2	3	11
204535	8	7	3	2	1	11
204534	8	7	3	2	2	11
204533	8	7	3	2	1	11
37469	8	7	5	4	100	11
204532	8	7	3	4	3	12
204525	8	5	4	3	1	10
204524	8	7	5	3	1	12
204537	3	1	3	2	20	10
204547	1	2	3	1	13	2
111332	1	4	1	2	2	4
111333	1	4	1	2	2	4
206325	1	3	1	1	13	15
153688	1	5	9	1	3	3
154145	1	8	1	1	2	1

Table 4.2: Copy number for each conserved element in each of the species

using the personal computer desktop architecture, obtaining the results in only just six hours. Even though it takes significantly a much longer time to process unassembled genomes, this is a remarkable performance improvement as scientists now are able to process unassembled genomes right at the time as genomes such as the African coelacanth (sequenced in 2013) [36] get sequenced. Such results would allow scientist to explore a much wider group or family of species, and ability to answer questions that in the past were impossible due to computational demands constraints.

Based on previous work [16, 17, 21], one of the biological questions that we wanted to answer is to find how far in evolution are we able to detect these extremely conserved regions between distant species. Our dataset consists of six species, three in which are chromosome assembled, and three species are chromosome unassembled **(Table 3.1)**. Because the evolutionary distance between the species is so great, there were no LIMEs found between rat and coelacanth fish, thus, we reduced the minimum length to only 50 base pairs. By using 50 base pairs as our minimum length, we decrease the amount of bases that get skipped, thus decreasing in performance since we explore twice the amount of data whereas in the previous dataset we used 100 base pairs as our minimum length. Stephen *et al* 2008, showed that a significance amount of conserved elements were common in tetrapods but not in the fish, indicating a slow down of the mutation molecular clock rate.

Given such phenomena, we wanted to see if we could detect what LIMEs are new in tetrapods, and what LIME elements would be considered only in the earlier fish. One of the challenges to answer these questions however, was computation time. Our dataset is composed of the following six species: Human, Rat, Coelacanth, Puffer fish, Elephant shark and Lamprey. Human, Rat and Puffer fish are the assembled genomes where Coelacanth, Elephant Shark and Lamprey are unassembled **(Table 3.0)**.

Between any of the pair species, we found approximately 2.1 million unique LIMEs in total, where approximately 57,000 of which are considered as complex and two million as simple LIMEs. The longest complex LIME sequence detected that is common to all species in our dataset has a length of 75-bp, where longest simple LIME has a length of 90 base pairs. On the contrary, the longest complex LIME between any of the species has a length of 861-bp where the shortest complex LIME is 59 base pairs (common between any two species). Simple longest LIME on the other hand has a length of 791 and whereas shortest LIME is 50 bp. There were 696 total LIMEs that are common to all six species, 51 of which were considered complex and 645 were considered simple - 38 unique without overlap. From a biological standpoint, scientists are more interested in complex LIMEs as they play important roles in functions such as regulatory mechanism. We found between any pair species (one against all), in human there were 52,902; mouse with 53,449; puffer fish with 1,825; coelacanth with 7,787, sea lamprey with 1,341 and last but not least elephant shark with 2,784. In addition, utilizing the 1000genome project [40], we found that 10,729 (non-redundant) transcription factors overlap or contain the 696 LIMEs that were common to all species in our dataset.

Of the 51 complex LIMEs common to all species, 13 are associated with protein coding genes and 38 are associated with different regulatory mechanism. Such functions include: Telomerase - a component of the CST complex which protects chromosome ends from degradation and end fusions [28, 29], Protein complex binding, Transporter activity, and Transporter mitochondria and synaptic vesicle precursors. In addition, the location distribution of the 51 complexes is widely diverse as they occupy different regions within or different chromosomes **(Table 3.1, Figure 4.0)**. One such element is associated with the KIF1B gene, associated with the Charcot-Marie-tooth neuropathy type 2a1 disease. This elements had 15 different copies in 10 different chromosomes in human, 86 copies located in all chromosomes in

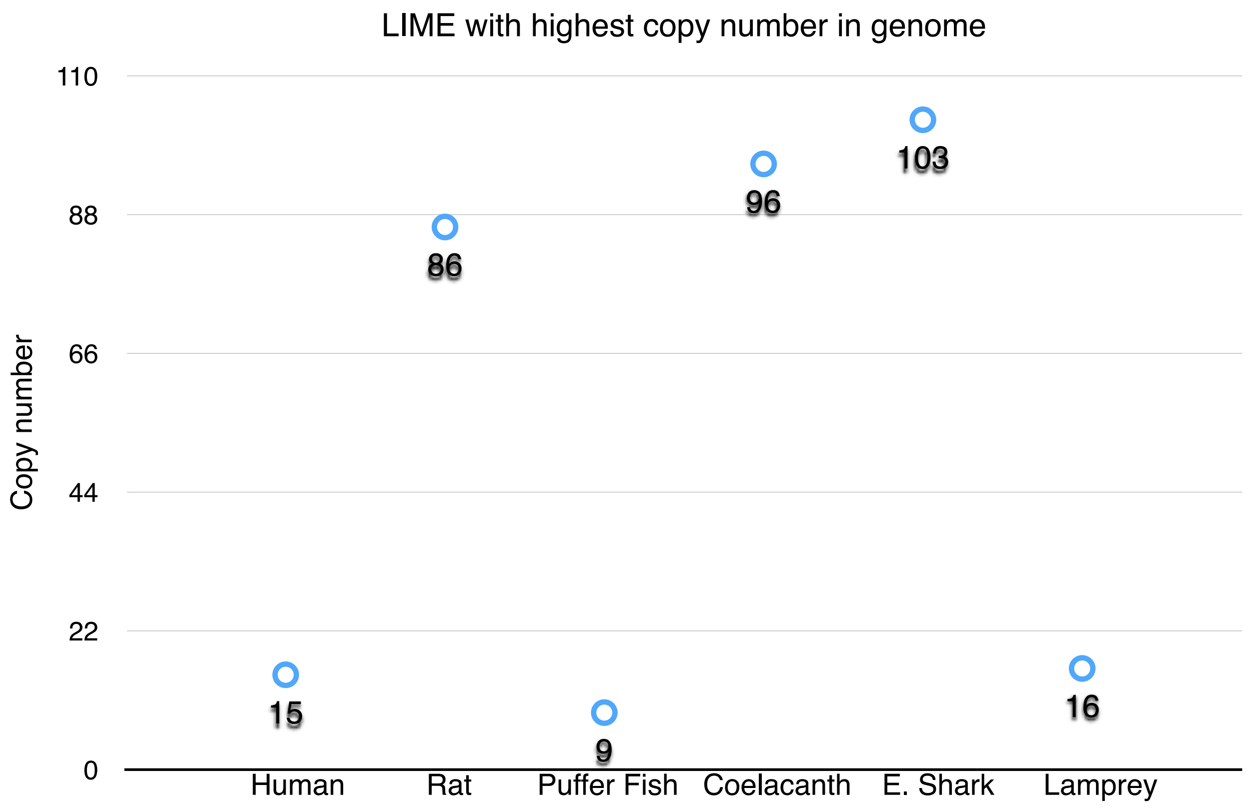

Figure 4.1: There are 51 LIMEs that are conserved among all 6 species. Here we show the highest number of copies a LIME occurs in the size species.

mouse, a single copy in puffer fish, 3 copies in coelacanth, 2 copies in elephant shark and 65 copies in lamprey **(Table 3.2).**

A significant amount of the total LIMEs belonged between Human and Mouse, with 1.2 million simple and 49,468 complex LIMEs (out of 2 million) respectively. Puffer fish, elephant shark and lamprey had 109 complex and 1,099 simple LIMEs. Adding the coelacanth fish to human and mouse, the number of LIMEs common to all three species dropped significantly, yielding 1,874 complex and 236,634 simple LIMEs. Knowing which LIMEs are present between human, mouse and coelacanth gives us the ability for find out exactly which elements are present and were preserved from the early fish to the tetrapods, as well as which elements disappeared, i.e. present in the earlier fish but not in the tetrapods. By taking the intersection between coelacanth and the earlier fish, there were 75 complex and 793 simple elements. Then, taking the intersection and set difference between human, mouse and coelacanth with puffer fish, elephant shark and lamprey, we found that there were 51 complex LIMEs that were preserved (which we expected) - where 24 elements that were present in the fish family were removed (or potentially mutated) in the tetrapods. There were 1,823 complex LIMEs that were gained in the coelacanth fish. Similarly, there were 148 and 235,989 simple elements in which were lost and gained (or born) respectively, i.e. present in the tetrapods but not in the earlier fish.

An interesting observation can be made in the amount of elements found between the tetrapods alone vs the earlier fish. We observe that most of the complex LIMEs are shared among the tetrapods where a fraction of the total LIMEs are common in the earlier fish. Based of this observation, we can postulate that the tetrapods are more similar (sequenced based) than the earlier fish, or, the evolutionary molecular mutation clock rate is much slower in tetrapods than in the early fish. The latter phenomena corresponds with [22] where it was concluded the molecular mutation clock rate during the earlier fish era was much faster than what we observe now in the tetrapods. Authors suggest that this phenomena makes sense due to the additional support in regulatory and developmental functions that the tetrapods require, thus a new (or evolved) set of elements are required for the survival of the organisms.

Chapter 5

Conclusion and future work

5.1 Summary

In conclusion, we have developed a hybrid alignment-fee hash-map, cache-oblivious approach that allows us to process almost any number of assembled species. More importantly, with our method, one can now process all of the assembled species to date utilizing as small as a personal desktop computer architecture. Additionally, even though assembled genomes are significantly less computationally demanding compared to unassembled genomes, utilizing current existing methods even for as small as two species, computation time becomes impractical to the average person. Ensembl [27] database has currently 66 species, approximately half of which are assembled. Processing such large number of species becomes impossible even having a cluster environment whereas it is impossible for one to process the same assembled species using a personal computer. As a consequence, our method has opened the door to a new set of species that the scientific community has not been able to process to date. In spite of plethora of sequencing data, approximately 33 and 700 unassembled species in the Ensembl [27] and Gold [32] databases respectively, we are still far from creating a complete encyclopedia of functional and structural elements of the genomes. Utilizing our new algorithm approach, we can provide a comprehensive atlas of the regions of extreme conservation in higher eukaryotes providing insights into the structural organization, function and evolution of these elements.

5.2 Future work

One of the drawbacks to our hybrid approach is the fact that our conserved element results are based on 100 percent identity between two sequences. While these extremely conserved elements give profound insights into the structural and functional evolution, our hybrid approach would miss any such conserved regions that have variations as small as a single nucleotide. This phenomena is in fact quite possible due to first, population variation, and second, single point mutation that organisms incur throughout their lifetime. Having this additional information allows us to further categorize these extremely conserved elements and give deeper understanding of evolutionary patterns that species may undergo. Thus, one of our goals is to extend the ability of our approach in order to account for highly conserved regions without using sequence alignment methods.

Performing analysis across all eukaryotic species would shed further insight into the patterns for highly and extremely conserved elements respectively. In additional, it is important to obtain a comprehensive annotated atlas between any number of species with respect to function, disease, novel and death of the elements as well as the genetic structural between species.

Bibliography

1. O'Driscoll *et al* 2013. 'Big data', Hadoop and cloud computing in genomics Journal of Biomedical Informatics , Volume 46 , Issue 5 , 774 - 781

2. Venter *et al* The Sequence of the Human Genome Science 291, 1304 (2001); DOI: 10.1126/science.1058040

3. Chial *et al* 2008. DNA sequencing technologies key to the Human Genome Project. Nature Education 1(1):219 (2008)

4. http://en.wikipedia.org/wiki/Whole_genome_sequencing#mediaviewer/File:Historic_cost_of_sequencing_a_human_genome.svg

5. Thijssen, Computational Physics, Cambridge University Press (2007). ISBN 0521833469.

6. Adebiyi *et al 2001.* An efficient algorithm for finding short approximate non-tandem repeats. Bioinformatics 2001, 17 Suppl 1:S5-S12.

7. Altschul *et al 1990*. Basic local alignment search tool. J Mol Biol 1990, 215(3):403-410.

8. Altschul *et al* 1997, Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 1997, 25(17):3389-3402.

9. Benson, Tandem repeats finder: a program to analyze DNA sequences. Nucleic Acids Res 1999, 27(2):573-580.

10. Castelo *et al* 2002. TROLL--tandem repeat occurrence locator. Bioinformatics 2002, 18(4):634-636.

11. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge, UK: Cambridge University Press; 1997.

12. Hauth *et al* 2002. Beyond tandem repeats: complex pattern structures and distant regions of similarity. Bioinformatics 2002, 18 Suppl 1:S31-37.

13. Kolpakov *et al* 2003. Efficient and flexible detection of tandem repeats in DNA. Nucleic Acids Res 2003, 31(13):3672-3678.

14. Landau et al : An algorithm for approximate tandem repeats. J Comput Biol 2001, 8(1):1-18.

15. Ning *et al* 2001. SSAHA: a fast search method for large DNA databases. Genome Res 2001, 11(10):1725-1729.

16. Stokes *et al* 2006. MICA: desktop software for comprehensive searching of DNA databases. BMC bioinformatics 2006, 7:427.

17. Reneker *et al* 2012. Long identical multispecies elements in plant and animal genomes PNAS 10.1073/pnas.1121356109.

18. Bejerano *et al* 2004. Ultraconserved elements in the human genome. Science 2004, 304(5675):1321-1325.

19. Waterston *et al* 2002. Nature 420, 520 (2002).

20. Roskin *et al* 2003. in Proceedings of the 7th Annual International Conference on Research in Computational Molecular Biology (ACM, New York, NY, 2003), pp. 257–266.

21. F. Chiaromonte *et al.*, Cold Spring Harbor Symp. Quant. Biol. 68, 245 (2003).

22. Stephen *et al* 2008. Large-scale appearance of ultraconserved elements in tetrapod genomes and slowdown of the molecular clock. Mol Biol Evol 2008, 25(2):402-408

23. Wang *et al* 2011. "A Fast Multiple Longest Common Subsequence (MLCS) Algorithm", IEEE Transactions on Knowledge and Data Engineering, 2011; Mar; vol. 23, no. 3: 321-33

24. Sankoff, "Matching Sequences Under Deletion/Insertion Constraints," Proc. Nat'l Academy of Sciences USA, vol. 69, pp. 4-6, 1972.

25. Smith and Waterman, "Identification of Common Molecular Subsequences," J. Molecular Biology, vol. 147, pp. 195-197, 1981.

26. Lipman *et al* 1985. "Rapid and sensitive protein similarity searches". Science 227 (4693): 1435–41. doi:10.1126/science.2983426. PMID 2983426.(1985)

27. Flicek *et al* 2013. Ensembl 2013 Nucleic Acids Research 2013 41 Database issue:D48-D55 doi: 10.1093/nar/gks1236

28. Wheeler *et al* 2005. Database resources of the National Center for Biotechnology Information. Nucleic Acids Res 2005, 33(Database issue):D39-45

29. "The computations were performed on the HPC resources at the University of Missouri Bioinformatics Consortium (UMBC)."

30. Jain *et al*, J. P. Telomeric strategies: means to an end. Annu. Rev. Genet. 44, 243–269 (2010)

31. De Lange, *et al* 2009. How telomeres solve the end-protection problem. Science 326, 948–952 (2009)

32. Pagani *et al*, The Genomes OnLine Database (GOLD) v.4: status of genomic and metagenomic projects and their associated metadata. NAR 40, D571-9.

33. Needleman *et al* 1970. "A general method applicable to the search for similarities in the amino acid sequence of two proteins". Journal of Molecular Biology 48 (3): 443–53. doi: 10.1016/0022-2836(70)90057-4. PMID 5420325. (1970)

34. Califano *et al* 1993. FLASH: a fast look-up algorithm for string homology. Proc Int Conf Intell Syst Mol Biol 1993, 1:56-64.

35. Glazov *et al* 2005. Ultraconserved elements in insect genomes: a highly conserved intronic sequence implicated in the control of homothorax mRNA splicing. Genome Res 2005, 15(6):800- 808.

36. Zheng *et al* 2008. Ultraconserved elements between the genomes of the plants Arabidopsis thaliana and rice. Journal of biomolecular structure & dynamics 2008, 26(1):1-8.

37. Camon *et al* 2004. The Gene Ontology Annotation (GOA) Database—an integrated resource of GO annotations to the UniProt Knowledgebase. In Silico Biol. 4:5–6. 2004

38. Consortium. 2006. The Gene Ontology (GO) project in 2006. Nucleic Acids Res. 34:D322–D326.

39. Hsu *et al* 2006. The UCSC known genes. Bioinformatics. 22:1036–1046. 2006

40. Amemiya *et al*, Nature 496, 311–316 (18 April 2013) doi:10.1038/nature12027

41. Durbin *et al* (2010). "A map of human genome variation from population-scale sequencing". Nature 467 (7319): 1061–1073. doi:10.1038/nature09534. PMC 3042601. PMID 20981092