Thesis for the Degree of Ph.D. in Science

Algorithms for comparing and visualizing genome-scale datasets

January 2013

Graduate School of Science and Technology
Keio University

Kristoffer Popendorf

主　論　文　要　旨

報告番号	㊙甲 乙 第　　号	氏 名	ポペンドフ　クリストファー

主論文題目:

Algorithms for comparing and visualizing genome-scale datasets

(ゲノム規模データセットの比較解析および視覚化するためのアルゴリズム)

（内容の要旨）

近年の超並列シークエンシング技術の発展によって、ゲノムに関するデータを新しく生成するコストと時間が急速に下がり、ゲノムを解読するプロジェクトの数が急激に増えている。現在2,547種の真核生物と12,460種の原核生物のゲノム解読プロジェクトが進行中である。また、ヒトやチンパンジー，マウスなどの57種の脊椎動物のゲノムがドラフト配列として完成されたので、ヒトに関連するゲノム解析を強力に推し進めることができる時代になった。しかし処理するべきデータの量と配列の数が指数的に増えているため、従来の解析手法をそのまま適用すると非現実的なコストがかかってしまう。

本論文では、ゲノム規模データセットを解析するための2つのアルゴリズムを提案する。第1のアルゴリズムは、複数のゲノムから相同領域を並列計算機で検索するためのアルゴリズムの構築である。第2のアルゴリズムは、近年開発された次世代シークエンサーから生成される大量のリードデータを解析した結果を視覚的に分かりやすい形で高速に表示するアルゴリズムである。

第1章では、ゲノム解読と比較ゲノム解析、次世代シークエンスデータとその解析および問題点について述べた。

第2章では、相同領域を検索するために開発したMurasakiと呼ばれるアルゴリズムについて述べた。複数の大規模な全ゲノムでも効率的に比較できるようにするために、計算機クラスターを用いた並列計算によって高速にゲノム比較が計算できる手法を開発した。並列計算の効率を保証するために、ハッシュ関数の計算および計算ジョブの振り分け方に関する新しいアルゴリズムを実装した。脊椎動物のゲノムデータを用いた検証実験により、相同領域の検出精度、並列計算の効率のいずれにおいても既存手法よりも優れていることを確認した。

第3章では、大量のシークエンスデータを解析した結果を高速に表示するアルゴリズムについて述べた。次世代シークエンサーから生成された何千万本の短いリード配列は、リードマッピングのプログラムを用いて参照ゲノムに配置される。配置されたリードマッピングの形状から生物学的な解析を行うためには、全ゲノム領域中の形状を高速に視覚化するためのソフトウエアが必要である。哺乳類規模の大きなゲノムに大量のリードデータをマッピングした場合、ゲノムレベルで俯瞰する表示から一塩基レベルでの詳細な表示までをスムーズに可視化するプログラムは存在しない。そこで、Samscopeと呼ばれるコンピュータグラフィックスの手法を取り入れた新しい表示プログラムを開発した。チキンゲノムのセントロメア解析において、Samscopeはタンパク質結合部位をゲノム配列から視覚的に発見することに威力を発揮することが示された。

第4章では、本研究を総括するとともに、提案した2つのアルゴリズムについて他のゲノム解析問題への応用可能性を議論した。

SUMMARY OF Ph.D. DISSERTATION

School	Student Identification Number	SURNAME, First name
School of Fundamental Science and Technology	80745137	POPENDORF, Kristoffer

Title

Algorithms for comparing and visualizing genome-scale datasets

Abstract

Recent years have seen a massive explosion in the number, complexity, and raw volume of new sequencing data thanks to advances in modern sequencing technology. In particular the advent of massively parallel sequencing, or colloquially "Next Generation Sequencing" or "NGS," has opened up a new world of sequencing applications that were once impractical at best. Whole bacteria can now sequenced in a matter of days, new mammalian genomes can be sequenced for a fraction of what they once cost, and well known species like *homo sapiens* can be re-sequenced to discover novel genetic variants for less than a $1000. In the first chapter of this dissertation, we review the current state of genomics sequencing technology, its applications, and current challenges.

One of the products of this sequencing explosion has been a wealth of newly sequenced genomes, including 57 vertebrates. With such rich data concerning some of our closest evolutionary relatives, comparative genomics studies promise to provide great insight into our physiology and development though analysis of similarities of whole genomes across multiple species. However, existing comparative genomics tools are capable of dealing with a few chromosomes at one time, and require excessive computational resources to keep pace with the vast number of genomes rapidly becoming available. To address this problem, we introduce a new approach to parallel sequence similarity search which offers efficient use of cluster computing resources to provide the scalability necessary to analyze current and future genome projects. We've named this algorithm Murasaki, and its details are described in Chapter 2.

Another application of NGS technology has been in areas of transcriptome, regulation, and variant analysis. Two relatively new applications unique to NGS use the massive number of reads available from NGS to assay RNA products by sequencing the RNA itself (RNA-Seq), or capture and sequence the DNA bound to specific transcription factors or DNA-binding proteins (ChIP-Seq). The data from these experiments can be hard to understand because of the scale of the data involved is overwhelming and requires some practical reduction to find features of interest before conducting a more detailed investigation. Existing techniques for visualizing NGS data has been limited to examining small regions and/or offered limited support RNA-Seq/ChIP-Seq features. To address this problem we propose a new algorithm and data format implemented in our program, Samscope, described in chapter 3.

In chapter 4 we summarize the impact of these new approaches, and examine their potential future areas of development.

Contents

Chapter 1

Introduction

1.1 Overview

Advances in robotics, chemistry, and imaging have allowed modern biology to gather more data more quickly ever before. In particular recent advances in sequencing technology have made genetic sequencing common place, to the point that unknown bacteria can be sequenced and assembled in a matter of days for a few thousand dollars (Schatz, Witkowski, and McCombie, 2012). These advances in sequencing technology have allowed the collection of available sequence data to grow exponentially. As of this writing in October 2012, the Genomes OnLine Database (GOLD) lists 3776 completed genomes (3595 prokaryotes and 181 eukaryotes), and 14710 new genome projects in progress (Pagani et al., 2012). As the number and scale of available sequences has increased, so too has the complexity of analyzing such large scale data. There are two primary aspects to this complexity:

1. The operational cost of running analysis algorithms over large genomic data at scale.

2. The task of allowing human researchers access to massive amounts of data to users through conventional computing resources (i.e. the keyboard, mouse, and monitor).

In this dissertation we present my work in addressing problems specific to these two aspects. In the first problem, the running cost of analysis of multiple large genomes, we consider the task of homology search across multiple genomes. Homology search is a fundamental problem in biology and the first step for a variety of tasks. Moore's law predicts an approximate doubling in the number of transistors on microprocessors every 2 years (Moore

et al., 1965). It is however an over simplification to conclude that Moore's law makes previous alignment algorithms for homology search sufficient for keeping pace with the exponential growth of sequence data. While the number of transistors per chip has kept pace with the expected exponential curve to date, the speed of individual CPUs has plateaued and has not increased in recent years now, due primarily to increasing thermal density on microprocessors the limits of thermal noise in modern semiconductor design (Kish, 2002). Modern processors have developed "horizontally," putting multiple processors ("cores") on the same chip, designed to function in parallel. Unfortunately, software design methods have not kept pace with the horizontal parallel growth of processors. Outside of the trivially parallelizable problems, parallel and concurrent software design remains a complicated world of individually crafted solutions to specific problems. Consequently the vast majority of algorithm design in the biology community is focused on serial (non-concurrent) algorithms. However as we approach the quantum mechanical limitations of individual processing units regarding size and thermal noise, it is critical that we develop new scalable approaches to use parallel computing architectures to solve biology problems. The approach we describe in this dissertation is a novel and efficient approach for running homology search in concurrently over commodity CPUs and networks.

The second problem is one of human interaction. In Chapter 3 we describe a novel approach for visualizing complex next-generation sequencing data across whole genomes at interactive rates. Modern displays can display images with dimensions of about a thousand pixels on a side (the current standard is "FullHD" at 1920 x 1080 pixels). This is a limit of both the display media and bandwidth necessary to push millions of pixels per second over a wire from the controlling computer. More importantly, on the human user end, at the most acute point of our visual field (the fovea) we can discern at most 50 cycles per degree (i.e. being able to discern line 100 µm thick lines at a distance of 50 cm) (Russ, 2007). The consequence of our limited visual acuity is that, even assuming optimal display medium technology, we would need a display 100 km long to draw a graph showing 1 data point for all 3 billion bases of the human genome (or about 500 km long using commodity PC displays). This is obviously impractical, so genome browsers historically have approached "genome display" problems with a button based "scroll and zoom" interface, as in the UCSC genome browser (Karolchik et al., 2003) and EBI's Ensembl (Hubbard et al., 2002). While appropriate for genes and relatively continuous features like "GC Content", this approach is awkward at best for analysis of modern data sources such as ChIP-SEQ and RNA-Seq, where per-base values are highly discontinuous. This is because while features like "GC Content" or gene content are reasonably summarized as a

single scalar feature like an "average" over 100 bases or even a million bases, the complex expression patterns in data sets such as ChIP-Seq and RNA-Seq are lost when reduced to a single value such as an "average".

In this chapter we'll provide a brief summary and background describing these two problems, and our approaches to address them.

1.2 Comparing Genomes

Continued developments in sequencing technology (Schatz, Witkowski, and McCombie, 2012) are driving an explosion in new sequencing projects in almost every taxa and increasing breaking traditional "taxonomy" boundaries with metagenomics projects, where DNA is collected from all the organisms in a given environment (e.g. a bucket of seawater, or a patient's lungs)(Pagani et al., 2012). For the vast maority of these projects, the raw DNA data means very little on its own, however it is very powerful in the context of other sequence data. The study of genomic sequences in the context of others can be broadly termed "comparative genomics."

1.2.1 Comparative Genomes

Comparative genomics is a broad term which can be made to encompass almost any study which involves comparing sequences from one genome to another. There are numerous biological applications to comparative genomics studies. For example, to better understand the evolutionary mechanisms of genomic rearrangement, researchers have examined the large scale genomic rearrangements between human and other genomes (the mouse genome in particular) (Alekseyev and Pevzner, 2007) looking for evidence of some genomic signals which might predispose some sites to rearrangement. Even the discovery that intrachromosomal rearrangements are more frequent than interchromosomal rearrangements (Pevzner and Tesler, 2003) was a significant breakthrough which required exhaustively searching for homologous regions of the human and mouse genomes.

Where previous studies have shown the involvement of certain genes in human development, comparative genomics analysis has been used to uncover new insights. For example, a mutation in the FOXP2 gene was previously shown to be related to a severe speech and language disorder (Fisher et al., 1998), but as advances in sequencing technology made possible the sequencing of FOXP2 in other closely related species, we now know that FOXP2 is very strongly conserved among most mammals, but humans show a unique mutation (Enard et al., 2002). Now that so many more complex genomes (in

particular vertebrates) have been sequenced, measures of "taxa-wide conservation" have become a commonly used feature for genomic analysis (Miller et al., 2007). Building on this technique, further studies looking at short regions highly conserved regions in mammals have used comparative genomics analysis to identify a series of "human accelerated regions" where humans show unusually high mutation rates compared to their primate relatives (Pollard et al., 2006). Studies such as this provide valuable insights not only into evolutionary mechanisms and methods to analyze genomic data for artifacts of evolutionary history, but also into what makes us essentially human. Now that so many more complex genomes (in particular vertebrates) have been sequenced, measures of "taxa-wide conservation" have become a commonly used feature for genomic analysis (Miller et al., 2007).

1.2.2 Homology search

Comparative genomics applications depend on the identification of regions of high similarity, and most with the stipulation that the similarity is the result of sharing a common ancestor. Sequences that share a common ancestor are by definition "homologous sequences" (Koonin, 2005) and as a consequence usually more similar to each other than unrelated genes or genomes. Because related sequences are expected to be similar, similarity itself is often used as representative of homology (usually assessed with some estimation of a p-value representing the probability that the sequences are similar by random chance rather than by common ancestry) (Altschul et al., 1990). Thus the task of finding likely homologs is usually well addressed by searching for "unusually similar" regions of sequence, and the terms "homology search" and "similarity search" are often used interchangeably (even if, strictly speaking, they have distinct different meanings). We use the term "homology search" here for simplicity and consistency with other "similarity search" literature.

As described above, the task of identifying likely homologous sequences requires the selection of subsequences which are so similar they're unlikely to have occurred by random chance. Given a sequence length n, there are trivially 2^n different subsequences we might select. It's obvious that trying each subsequence in turn would be a very slow approach, and in fact finding the longest common subsequence in the general case is an NP-Complete problem (Maier, 1978). However given an alphabet of constant size and constant number of sequences (as is the case for genomes), the ubiquitous Smith-Waterman algorithm (Smith and Waterman, 1981) provides a polynomial time solution to find all optimal scoring local alignments. Here a local alignment implies a selection of insertion/deletion/mutation operations relating the sequences under consideration. However, the time and space required for

Smith-Waterman is $O(N^S)$ where S is the number of number of sequences and N is length of the longest input sequence. Because of the sensitivity achieved by considering all possible alignments, there has been interest in producing FPGA (Li, Shum, and Truong, 2007) and ASIC (Han and Parameswaran, 2002) implementations of Smith-Wateman, these implementations accelerate the Smith-Waterman each process but do not solve the fundamental the $O(N^S)$ scaling problem. The general consensus is that comparing more than 2 sequences at once or sequences longer than a few kilobases requires a heuristic to limit the search space to a subset of likely optimal alignments.

1.2.3 Murasaki and Past Work

Starting with FASTA (Pearson and Lipman, 1988) these heuristics have generally focused on some series of exact-matching bases. FASTA and many algorithms after it build a look-up table of k-mers. K-mers are k length strings from the alphabet of sequences under investigation, Σ. In FASTA and algorithms like BLAST after it (Altschul et al., 1990), more sensitive alignments are only considered in the subsequences surrounding a matching k-mer (thus ensuring a certain degree of similarity before exploring a more expensive calculation). A complete lookup table to contain all possible k-mer seeds thus includes Σ^k entries. In the literature, this lookup table gets called hash table, however typically in practice no "hashing" in the traditional sense is involved (or the hash function is an identity function $H(x) = x$), as in practice all k-mers are expected to appear somewhere in the sequence for small values of k. A matching set of subsequences of sufficient similarity to trigger a more expensive alignment is called an "anchor." Using larger values for k increase the minimal level of similarity required; this can be thought of as trading sensitivity for speed. In general the subsequences that are used to locate anchors are termed "seeds," though so far we've only considered k-mers. PatternHunter (Ma, Tromp, and Li, 2002) introduced the idea of using "spaced seeds," where specific bases in the seed are not required to match to increase sensitivity while requiring the same number of matching bases. This strategy is easily implemented in place of existing k-mer tables, and was soon implemented in BLASTZ (Schwartz et al., 2003), which was in use for pairwise whole genome comparison. All of the above algorithms however consider only *pairwise alignment* and do not directly address multiple alignment. TBA implemented a multiple alignment algorithms on top of BLASTZ (Blanchette et al., 2004) using all $\frac{(S-2)(S-1)}{2}$ (or $O(S^2)$) pairwise alignments of a set of S sequences. When the total length of all input sequences is N, this allows the calculation of anchors in S^2N time. ROAST (Miller et al., 2007) provided an additional short-cutting heuristic to the MULTIZ align-

ment algorithm part of TBA, where a guide tree could be used to align all $s_1, s_2, \cdots S_{n-1}$ sequences to one reference sequence s_n, thus ostensibly reducing the cost for anchoring where a guide tree is already known to $O(SN)$ rather than $O(S^2N)$ with TBA. However calculating a guide tree through ordinary methods will again require $O(S^2N)$ time (Thompson, Higgins, and Gibson, 1994).

Murasaki provides an alternative way of anchoring multiple large sequences without relying on guide trees or calculating $O(S^2)$ pairwise alignments. The core idea for doing this is to keep all sequences under consideration in the hash table. This greatly increases the size of the hash table, therefore Murasaki implements a variety of features to increase the storage efficiency of its hash table. Furthermore Murasaki introduces a novel method for parallelization of anchoring by distributing the task of calculating, storing, and ultimately computing anchors from the hash table across a computer cluster. Great care was given to implement a distributed algorithm using MPI to achieve near constant and balanced use of resources, ultimately achieving extremely high efficiency. The process is described in detail in Chapter 2.

1.3 Short read visualization

As we describe in Section 1.3.1, the new generation of massively parallel sequencing technologies has enabled millions of reads to be generated for a fraction of what they used to cost. This massive influx of new data has lead to a variety of new biological applications, but also introduced new bioinformatics challenges to making sense of the data. When trying to interpret new data, and check for unusual occurrences, traditional analysis tools designed for a relatively small number of long reads produced with capillary technologies are generally impractical. The vast scale of the data from massively parallel sequencers must be distilled into human digestible chunks, and there have been a number of applications which do so in a visual fashion, such as SAMtools (Li et al., 2009), Tablet (Milne et al., 2010), and the "Integrative genomics viewer" (IGV) (Robinson et al., 2011). We introduce a new visualization tool, Samscope, which allows users a highly responsive view of a wide variety of statistics of their data in an easily comparable and extensible format.

1.3.1 Massively parallel sequencers

The main propellant behind the current sequence explosion is relative affordability of sequencing using the latest massively parallel sequencing tech-

nologies (Rogers and Venter, 2005). Rather than feeding individually prepared and amplified samples through individual capillaries as with traditional Sanger sequencing (Sanger, Coulson, et al., 1975), massively parallel sequencing technologies amplify millions of DNA templates simultaneously, and read them without manipulating individual samples. This dramatically reduces the cost of sequencing while vastly increasing the total throughput of a single sequencing machine.

There are many variations of massively parallel sequencing technology available, and currently each has carved out some niche market for which they are still currently the best for certain applications. Future developments from companies like Pacific Bio and nanopore sequencers promise new capabilities which could supersede some of the current technologies, but so far the major technologies are: Illumina's Genome Analyzer and HiSeq series, Roche's 454, Life Technologies' SOLiD and Ion Torrent sequencer technologies (Glenn, 2011). Despite their varying technologies and varying release dates, all of these new sequencing technologies are typically labeled "Next-generation sequencers" (NGS) in the literature, begging the question of where to draw generational lines. Some claim that "2nd generation sequencers" should include all of the above technologies, and "3rd generation sequencers" should include single molecule sequencing technologies, like PacBio and and HeliScope (Glenn, 2011). But for the purpose of this discussion, the important factor linking all of them is that they produce millions of short reads.

The new massively parallel nature of these sequencers has enabled new sequencing protocols for new biological inquiries. We describe a few them here for reference.

1.3.2 Short Read Mapping

Outside of de-novo sequencing, where the source genome for the short reads being produced is unknown, the origin of the reads under examination is typically known. An assembled reference genome sequence is used to align reads to the origin genome. After reads are aligned to the reference genome, various features can be calculated, such as variances between the sample and reference genomes, or the relative abundance of different genome sequences in the sample. As described in Section 1.2.2, alignment is a hard problem, and many variations of short read alignment programs have emerged, such as SHRiMP (Rumble et al., 2009), BWA (Li and Durbin, 2009), and Bowtie (Langmead et al., 2009). Fortunately a single format, SAM (*Sequence Alignment/Map*) and its compressed binary cousin BAM (Li et al., 2009), have emerged as the *lingua franca* of short read mapping data. This common language has allowed bioinformaticians to build complex yet portable

analysis and visualization tools and pipelines around a common file format.

1.3.3 DNA-Seq

DNA-Seq is the traditional "genome sequencing" task where a researcher seeks to find out the DNA content and structure of a sample. When using NGS, samples are typically fragmented into short strands a few hundred bases in length. Now most sequencers offer some form paired end sequencing, allowing researchers to sequence both ends of the fragment. Using this "paired end" data, the distance between the reads can be inferred to aid in de novo assembly (Pevzner and Tang, 2001; Zerbino and Birney, 2008), or in detection of structural variants (Tuzun et al., 2005). There are numerous variations on the DNA preparation technique (Teer et al., 2010), for example enriching only the DNA fragments which belong to known gene exons or matching some other user defined template. These enrichment techniques are typically used to increase coverage and consequently sensitivity over a specific genomic domain of interest, for example for detecting rare variants in a highly heterogeneous sample.

When visualizing the results of DNA-Seq runs, users often want to check for areas of unusually low or high coverage, examine areas with unusually high mutation rates or rearrangements, and look for clues that might indicate the origin of the unusual result, such as strand or G/C bias.

1.3.4 RNA-Seq

RNA-Seq is analogous to expressed sequence tags (EST) or serial analysis of gene expression (SAGE) in traditional sequencing, however many RNA fragments are reported in parallel. Typically mRNA is fragmented, PCR amplified, and converted to cDNA, after which it can be prepared and read in the same fashion as DNA-Seq (Mortazavi et al., 2008).

RNA-Seq poses a different challenge to visualize in that a large range of different expression values are expected, and across large swaths of genome most expression values will be at or near 0.

1.3.5 ChIP-Seq

ChIP-Seq or "Chromatin immunoprecipitation followed by sequencing" uses protein-antibody interaction to capture DNA fragments bound to a protein of interest (POI) (Park, 2009). Transcription factors and their binding sites have been the focus of numerous gene regulatory network studies for years (Farnham, 2009), however until recently it has been hard to experimentally

characterize the binding sites of DNA-interacting proteins. ChIP-Seq makes it possible to map the binding sites of a given protein across an entire genome in a single experiment. The typical protocol involves crosslinking proteins to the DNA using a reagent such as formaldehyde, then shearing the DNA into short fragments (200-600bp in length). Antibodies bound to a substrate are used to collect only those fragments which are bound to the POI, while the other fragments are washed away. The crosslinks are reversed, releasing the protein, and the remaining DNA can be sequenced as in DNA-Seq.

An interesting side effect of the ChIP-Seq process is that because only a single strand of DNA bound by the original POI is retained, the resulting reads are thus easily retained as strand specific. Furthermore, as the original binding site of the POI can be assumed to be present on all DNA fragments (and expected to be closer to the middle than the ends), the resulting read coverage is a bimodal distribution with reads on the "forward" strand clustered on one side, and the "reverse complement" strand on the other (Park, 2009). Subtracting the "reverse coverage" from the "forward coverage" yields a "read polarity" value. Binding sites can be inferred in areas of high coverage to occur around the polarity "zero-crossing point." Samscope makes visualization of these types of characteristics simple and intuitive. See Chapter 3 for details.

Chapter 2

Murasaki: A Fast, Parallelizable Algorithm to Find Anchors from Multiple Genomes

2.1 Introduction

"Homology search" plays a fundamental role in a variety of sequence analysis studies. The goal of a homology search is usually some form of the *Longest Common Subsequence* (LCS) problem. In the most general form, with an unbounded number of sequences, LCS is an NP-Complete problem, therefore any attempts to solve the problem quickly and at scale are forced to recognize only a limited subset of the problem. Limiting the number of sequences under comparison to some fixed number N allows the now ubiquitous Smith-Waterman polynomial-time dynamic programming solution (Smith and Waterman, 1981) to be used. Like most NP-Complete problems, what is easy for a few small objects becomes impractical for larger more numerous objects. Indeed the time and space requirements of Smith-Waterman are considered prohibitive for large (or more critically numerous) sequences, leading to the evolution of modern homology search algorithms that employ some heuristic to provide an approximation of the exact LCS solution. Newer algorithms like FASTA (Pearson and Lipman, 1988) and later BLAST (Altschul et al., 1990) and its derivatives (PatternHunter, BLASTZ, Mauve, etc.) rely on subsequences of unusually high conservation to "anchor" a search to a smaller area where a more detailed homology search can be conducted in reasonable time, from which the term *anchor* is derived. The increasing availability of sequences and the now common need to align multiple whole genomes has repeatedly pushed each of these homology search algorithms to the point where

they are no longer viable, demanding the development of software that takes advantage of new technologies and novel algorithms with refined heuristics. Our software, Murasaki, is yet another entry in this tradition. We also follow the UNIX tradition of making a tool to do one job and do it well. Thus we confine the scope of Murasaki to that of *anchor search* on *multiple genomes* only, and leave the question of what to do with anchors to other tools further down the toolchain. Our goal was to create an efficient flexible way to search for anchors that meet arbitrary constraints across multiple genomes (as opposed to simple pairwise comparisons) while taking advantage of the increasingly multi-core and distributed computational environments available to researchers.

2.1.1 Anchoring

The term "anchor" generally refers to well-conserved short regions among two or more genomes, and is biologically defined as a short gene-coding region or an exon in a long gene or non-coding region (including functional RNAs) where no rearrangement occurs. Computationally, anchors are generally determined by identifying occurrences of matching *k-mers* and extending or combining them as high scoring pairs.

The task of finding anchors is considerably different from producing alignments. Finding anchors is only the first step of BLAST (Altschul et al., 1990), in that BLAST produces lots of anchoring pairs and tries to extend them. Mauve (Darling et al., 2004), for example, relies on anchors only for finding the endpoints of alignable collinear regions. Therefore, anchoring alone is not expected to be as sensitive as exhaustive gapped-alignment, but anchoring multiple genomes can rapidly yield information that can be used to reduce the computation time of multiple genome alignments (Brudno et al., 2003), to infer genome rearrangements through synteny identification (Bourque and Pevzner, 2002), to find conserved non-coding RNA regions which are usually much shorter than protein-coding regions, and to execute genome-wide evolution analysis such as the identification of ultraconserved regions (Bejerano et al., 2004).

2.1.2 Previous Work

Modern homology search programs generally rely on some efficiently searchable data structure to index the locations of short subsequences (we will call these subsequences "seeds"). There have been many approaches to doing this. Mauve (Darling et al., 2004) uses a sorted list that is simple and space efficient, and because Mauve prunes all but the unique seeds, usually fast.

MUMmer (Delcher et al., 1999) and later ramaco (Ohlebusch and Kurtz, 2008) use suffix trees to find short exact matches. The latter implements a pairwise comparison based approach to finding matches across multiple sequences while relaxing the "unique" constraint of *multiMUMs*, however offers little opportunity for parallelization and is limited by the space requirements of its tree structures. The speed gains from FASTA/BLAST and the vast majority of popular modern derivatives such as BLASTZ (Schwartz et al., 2003) come from storing the seed index in a hash table where look-up of a given seed is constant time. In practice this hash table is generally a block of contiguous memory in the computer, such that we might think of it as a table of M entries, $T_0, T_1, T_2, \ldots T_M$. Key-value pairs (K, V) might then be recorded in the table by storing V in the entry T_i specified by a *hashing function* $H(K)$ (i.e., where the *hash* i of K is defined by $i = H(K)$). For homology-finding, the key K would generally be a "seed" (e.g., ATGC), and the value V would be the location in the input sequence(s) at which it occurs. Because ATGC might occur any number of times, hash table entries are often some list-like data-structure that allows a different V to be stored for each incidence of the same seed K. The performance of a hash table then depends on the ability to find the entries that match a given key quickly. In other words if $H(K)$ is slow, or storing to and retrieving from T_i is slow, performance deteriorates. Ideally $H(K)$ produces a different hash T_i for every different value of K, but when two keys K_i and K_j such that $K_i \neq K_j$ produce the same hash (i.e., $H(K_i) = H(K_j)$) separating their values V_i and V_j in the hash table requires additional work. These events are called "collisions." Thus to minimize the time spent resolving collisions, the selection of a hash function H that avoids collisions is at least as important as how to resolve them. In cases where the maximum number of keys is small, as in PatternHunter and BLASTZ where keys are at most 12 or 14 bases (limiting the number of possible keys to 4^{12} or 4^{14} respectively), the size of the hash table M can be chosen to accommodate all possible keys, and the hash function $H(K_i)$ can simply be the position of K_i in an enumerated list of all possible values of K (if we think of a string of nucleotides as a base 4 number, thus $H(K_i)$ becomes the trivial identity function $H(K_i) = K_i$). This is the standard method used by most existing hash-based homology search algorithms, and is acceptable for a small number of keys. However the size of the hash table required to guarantee no collisions increases exponentially with the length of keys (e.g., when using longer *k-mers*). Given 14 bases alone requires 2.68×10^8 entries, which at even a modest 32 bits per entry is 1GB of memory, 15 bases requires then 1.07×10^9 entries and 4GB, 19 bases requires 2.7×10^{11} entries and 1TB and so on, it's obvious that if one wants to use longer keys a different solution is required. BLAST and BLASTZ limit

this exponential expansion by using only the first N bases as a key when the key length exceeds a predefined threshold.

Ma *et al.* introduced the notion of *spaced seed patterns* to homology-search in PatternHunter (Ma, Tromp, and Li, 2002). Spaced seed patterns are typically represented as a string of 1s and 0s, where 1s represent bases that contribute to a "seed" and 0s do not. For example given the pattern 1011 and the sequence "`ATGC`", we could generate two seeds, "`A.GC`" and "`G.AT`" (the reverse complement) where the "`.`" (period) characters are disregarded or can be thought of as matching anything, as in regular expressions. The *weight* of a pattern refers to the number of 1s in the pattern. Ma *et al.* showed that a spaced seed pattern is more sensitive to weak similarities than a non-spaced seed pattern of the same weight, leading to a small revolution in homology-search as programs were modified to incorporate spaced seeds. Calculating so called "optimal seed patterns" becomes a challenge for long seeds (Preparata, Zhang, and Choi, 2005). In general, however, shorter and lighter patterns are expected to be more sensitive while longer and heavier patterns are expected to increase specificity. The use of spaced seeds complicates the generation of hash functions, often limiting the choice of patterns (for example, BLASTZ offers users the choice of 2 spaced seed patterns).

When the MegaBLAST and BLASTZ approach of hashing using only the first N bases is applied to spaced seed patterns, we refer to this as the *First-N* approach. To examine the situations in which this *First-N* approach is suboptimal, considering genome hashing from a Shannon entropy perspective is helpful. Because a weight w seed K_i has at most w random symbols from an alphabet of {A,C,G,T}, K_i has at most $2w$ bits of information. The naive *First-N* approach to getting an h bit hash out of a $2w$ bit seed is simply to start reading bits from one end and stop once h bits are collected, as described above for BLAST and BLASTZ. If each base were statistically independent, any sampling of h bits from the key K_i would be equally effective. In reality, however, each base of a *k-mer* is far from statistically independent. In fact, the average conditional entropy of a base genome given the previous bases is estimated to be closer to 1 bit per base (Tabus and Korodi, 2008)(Farach et al., 1995). Therefore, the naive approach is expected to provide poor utilization of the available hash key space. We confirm this in **Results** (Chapter 2.3). At the other end of the complexity spectrum, we can expect near uniform utilization of the hash key space by passing all $2w$ bits to a cryptographically secure pseudorandom hash function like SHA-1 (National Institute of Standards and Technology (NIST), 2002) or MD5 (Rivest, 1992). SHA-1 and MD5 are often used as hash algorithms where the characteristics of the key domain are unknown and uniform utilization of the hash range is critical (such as in file systems (Quinlan and Dorward, 2002) to prevent data

loss). MD5 has recently been shown to be vulnerable to a variety of cryptanalysis attacks designed to generate colliding keys for a given key rendering it unsuitable for security purposes; however MD5 is faster than SHA-1 and the cryptanalytic attacks are irrelevant to our purposes here. These cryptographic hash functions are, however, computationally expensive and produce 256 bit hashes from which we can use only a small fraction. In Murasaki we introduce a novel hash function generation algorithm to automatically generate hash functions from arbitrary spaced seed patterns that approximate maximal hash key space utilization in a computationally inexpensive manner, which we term the "adaptive hash algorithm." The details of this algorithm are explained in Section 2.2.

2.1.3 Motivation

Identification of anchors (or seeds for alignment) for whole-genome comparison plays a fundamental role in comparative genomic analyses because it is required to compute genome-scale multiple alignments (Waterston et al., 2002; Dewey et al., 2006; Gibbs et al., 2004), and to infer among multiple genomes orthologous genomic segments descendended from the common ancestor without any rearrangement (Hachiya et al., 2009). A common approach for the identification of anchors among multiple sequences, used by TBA (Blanchette et al., 2004), first detects anchors between every pair of sequences, and then progressively integrates pairwise anchors to form anchors across multiple sequences. For a given number of sequences (N), this approach requires ${}_NC_2$ computations of pairwise anchors. Naturally this *progressive approach* requires quadratic ($\mathbf{O}(N^2)$) time with respect to the number of sequences. Linear time variations on this approach exist when alignments to a single reference sequence are appropriate (eg. the UCSC human conservation track (Miller et al., 2007)), however these have their own limitations which we describe in Section 2.4.1. Current progress in sequencing technologies accelerates the accumulation of completely sequenced genomes: 1,139 Prokaryotic and 129 Eukaryotic genomes are now available as of this writing (6th May, 2010) according to the GOLD database (Liolios et al., 2008). The rapidly increasing number of available genomes poses a scalability challenge for bioinformatics tools where computational cost is bound to the number of sequences. Progressive alignment is further complicated by the potential to introduce errors or bias based on the phylogenetic trees selected for progressive alignment and accumulate pairwise errors at each stage of the alignment (Kemena and Notredame, 2009). To address these issues, we propose an alternative to the progressive approach allowing the identification of multi-sequence anchors simultaneously wherein all sequences are

hashed simultaneously and well-conserved anchors are computed in a single pass. This allows us to compute anchors across multiple genomes with an approximately linear cost without any pairwise comparisons or tree inference.

2.1.4 Parallelization

With processor densities now pushing the constraints of physics for speed (Kish, 2002), chip manufacturers have abandoned increasing clock speeds in favor of adding multiple cores and increased parallelism. To deal with the exponentially increasing amount of sequence data available from new sequencing technologies (Bentley et al., 2008), new algorithms need to be designed to take advantage of multicore and cluster computing environments in order to keep up. Furthermore, conventional computer architectures impose a strict limit on the amount of maximum amount of RAM usable in a single machine. This has pushed developers working on whole genome data, as with ABySS (Simpson et al., 2009), to use cluster computing to avoid memory barriers even if there is little gain in computational speed (or even a decline).

When using progressive alignment tools such as BLASTZ and TBA, the ${}_NC_2$ comparisons for each pair of sequences are independent and therefore trivially parallelizable. Any finer grained parallelization (necessary to use more than ${}_NC_2$ processors), requires breaking the n sequences into smaller fragments that can be aligned independently (the technique used for Human and Mouse genomes in (Schwartz et al., 2003)). This fragmentation is in fact necessary with software like BLASTZ for mammalian scale genomes where the genome as a whole is too large to be processed by the alignment software in a single pass. Breaking each sequence into M fragments incurs an additional cost for each pair $\{S_i, S_j\}$ of the ${}_NC_2$ comparisons. With fragmentation each comparison is shorter, however each base is considered at least M times more than it was without fragmentation. This is because all M fragments of sequence S_i must be compared with all M fragments of sequence S_j and each fragment gets re-indexed and anchored each time. In Murasaki we eliminate both the ${}_NC_2$ and fragmentation costs by introducing a novel fine grained yet highly efficient approach to parallel anchoring using an unlimited number of processors independent of the number of sequences or fragments under comparison.

2.2 Implementation

2.2.1 Algorithm Outline

At its most primitive definition, Murasaki takes as input a set of DNA sequences, a *spaced seed pattern*, and provides as output a series of *anchors*.

Anchors are defined in Murasaki as a set of intervals across some subset of the input sequences. Each anchor contains at least one set of matching *seeds*. Here a *seed* refers to an input substring when masked by the *spaced seed pattern*. When an anchor is initially constructed based on a set of matching seeds, both ends are extended by an ungapped alignment until the minimum pairwise score falls below the X-dropoff parameter as in BLAST and BLASTZ (Schwartz et al., 2003). Overlapping collinear anchors are coalesced to form larger anchors, as in Figure 2.1.

Because our goal is to find and extend matching seeds, the role of the hash table is to accelerate the identification of matching seed sets. FASTA, BLAST, and BLASTZ all rely on hash table-like indices to find matching seeds in constant time. Mauve uses a "sorted k-mer list" where *k-mers* (or in later versions pattern masked *k-mers*) are stored in a list and sorted. Suffix trees and other tree-based approaches use some tree-like structure to accomplish the sorted index task. As described in **Previous Work** (Section 2.1.2), the strict one-to-one hash table-like approaches in the FASTA derivatives limit the size of seeds to $\log_4 M$ where M is the size of the hash table. Murasaki uses a hybrid approach mixing hash tables with a fast comparison based collision resolution mechanism to reduce the number of comparisons needed to find matching seed sets. Hashes are generated from seeds such that if two seeds K_i and K_j match, they necessarily produce the same hash (i.e., $K_i = K_j \Rightarrow H(K_i) = H(K_j)$), therefore all matching seeds will reside in the same location within the hash table. Collisions are resolved by either using chaining and a sort or by open addressing.

The algorithm is as follows:

1. Load the input sequences as 2-bit codes.
2. Determine hash parameters and hash function H.
3. For each location L across all input sequences (on both forward and antisense strands):
   (a) Compute a hash h for the seed K at location L based on the input sequence and hash function H
   (b) Store the pair (K, L) into location $H(K)$ in the hash table.

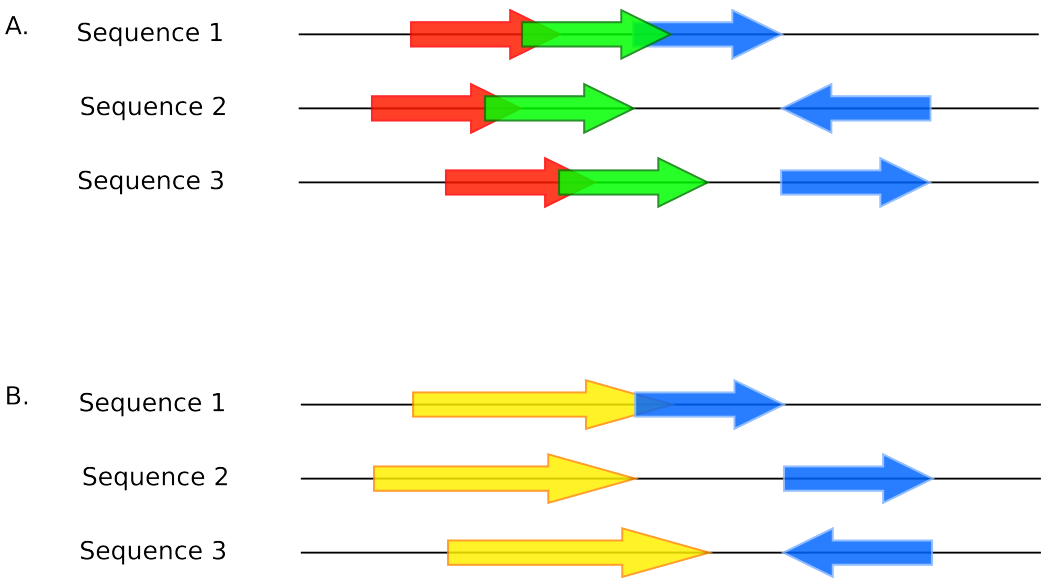

Figure 2.1: Here we illustrate an example of how anchor coalescing is processed. In **A** we show 3 anchors spanning 3 sequences represented by 3 sets of arrows: red, green, and blue. The green anchor overlaps the red anchor in all sequences, and maintains colinearity, therefore they can be coalesced. However the overlap of green and blue occurs only in Sequence 1, therefore they cannot be coalesced. **B** shows the results of the coalescing of green and red with the resulting anchor shown in yellow.

At this point all matching locations ("seeds") share the same hash key and therefore share the same locations in the hash table.

4. Extract all matching sets of seeds from each entry of the hash table (i.e., "invert" the hash table).

5. For each set of matching seeds:

   (a) Make a new anchor A for each subset of seeds such that there is exactly one seed from each sequence.

   (b) Extend each new anchor A by ungapped alignment.

   (c) Coalesce each new anchor with pre-existing existing anchors.

The hash table inversion and anchor generation steps are illustrated in Figure 2.2. Murasaki optionally supports partial matches, also known as "islands" where some number of sequences may be missing. In this case L for sequences up to the specified number of missing sequences are considered anchored at a special $\emptyset$ location.

It is worth noting at this point that the size of the hash table is a critical factor. Our hash table size is defined to be exactly 2^b where b is the "hashbits" parameter, describing the number of bits expressed in hash values. The events where $H(v_i) = H(v_j)$ and $v_i \neq v_j$ are termed "hash collisions." While careful selection of a hash function can reduce the number of hash collisions, the pigeon hole principle guarantees that some collisions must occur if the number of distinct seeds is greater than the size of the hash table. Even given a perfectly balanced hash function, where a seed selected at random has an equal probability of mapping to any key, the expected number of collisions per key is $\frac{4^w}{2^b} = \frac{2^{2w}}{2^b} = 2^{2w-b}$, where w is the weight of the pattern. Therefore, increasing the value of b by one is expected to reduce the number of collisions by half, dramatically reducing the time required to invert the hash table and extract matching seeds. This trade-off of memory for speed is common in hash tables and in data structures overall, but Murasaki is the only existing hash-based anchoring algorithm to separate the selection of spaced seed patterns from the data structure used to index the input sequences. This gives the user separate tunable parameters that allow control of sensitivity/specificity independent of the memory footprint on the system.

2.2.2 Parallelization

Murasaki's approach lends itself to parallelization at several points. First, the order in which individual seeds are hashed is irrelevant, and therefore we

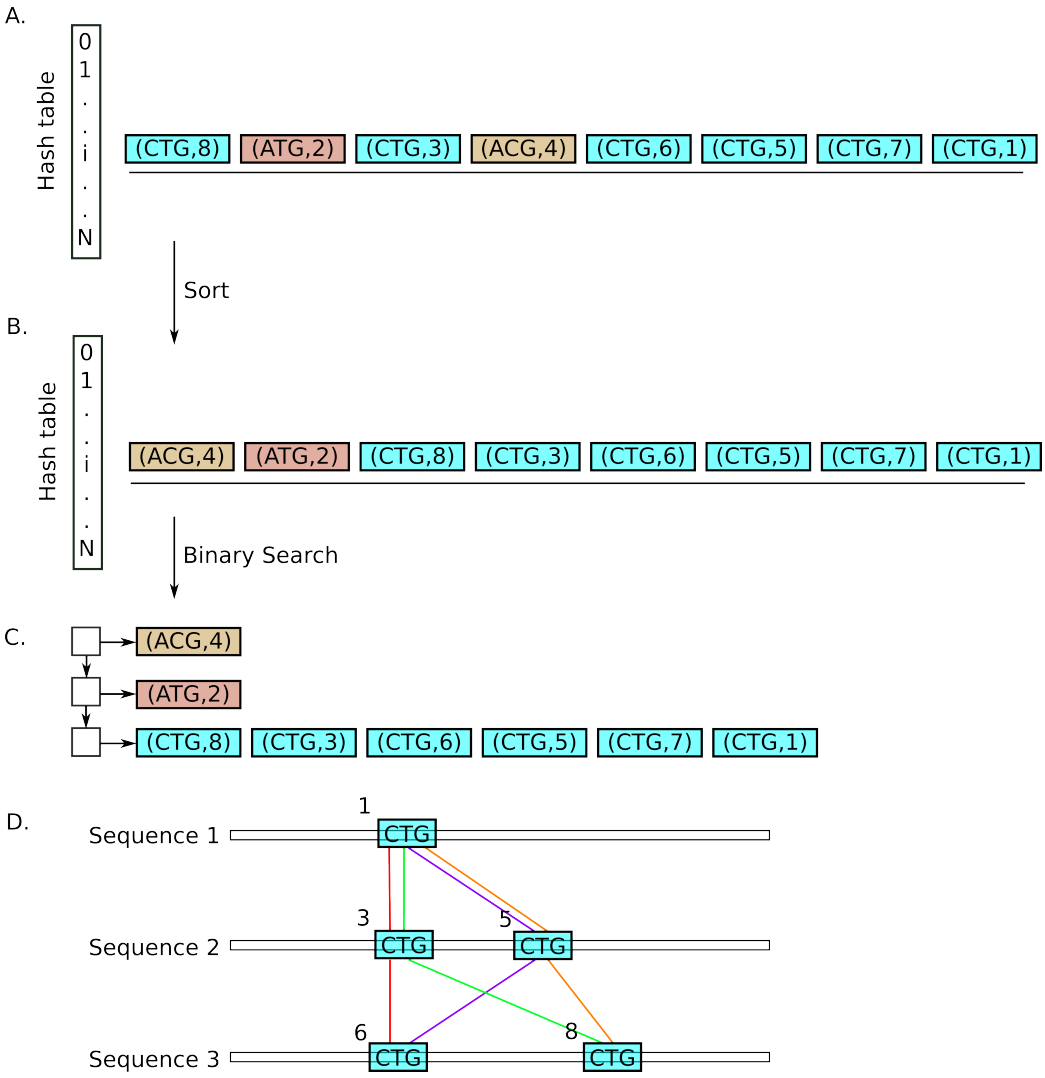

Figure 2.2: Here we show a simplified example of how matching seed sets are extracted from the hash table and converted into anchors. **A** shows one row (T_i out of $T_0 \dots T_N$) of the hash table. Several (K, V) pairs have been inserted into the hash table. V indicates a position in the input sequences at which K occurs. Because K is necessarily implied by V Murasaki only stores V, however here both K and V are shown for clarity. Different values of K have also been colored differently to note their difference. First, this row is sorted with the result shown in **B**. The extents of each matching seed set can then be found in $O(\log N)$ time by binary search. These matching seed sets are extracted into a series of lists, as shown in **C**, which are then used to construct anchors, as shown in **D**.

can devote as many CPUs to hashing as desired. The storage of locations into the hash table may require traversing and updating some form of list or tree structure, and which takes time comparable to that of computing a hash. Only one CPU can modify a list or tree at a time; however, if the lists are independent, CPUs can work on independent lists or without risk of interfering with each other. "Inverting" entries in the hash table can also occur in any order, and the more CPUs we apply to this task the faster it will finish. Therefore we divide all available computational nodes into one of two disjoint sets: "hasher nodes" and "storage nodes." These nodes function in a "producer/consumer" model where one set performs one half of an operation and passes the result to a node in the opposite set. Fundamentally the parallel algorithm works as follows:

1. All nodes load input sequences as 2-bit codes.
2. Hash parameters and a hash function H are generated.
3. Nodes are assigned a job as either "hasher" or "storage."
4. The input sequence is divided into contiguous segments, one for each hasher.
5. Storage nodes are assigned a contiguous interval of the hash table to manage.
6. Each hasher node
   (a) computes a hash h for the seed K at location L based on the hash function $H(K)$.
   (b) sends this (K, L) pair to the storage node responsible for K.
7. Meanwhile, each storage node
   (a) receives a (K, L) pair from a hasher node.
   (b) stores L into location h within the hash table.
8. Hasher and storage nodes now switch roles, the storage nodes becoming producers, and the hasher nodes becoming consumers.
9. Each storage node
   (a) inverts one row of the hash table at a time.
   (b) sends the each resulting set of matching seeds to an arbitrary hasher node.

10. Meanwhile each hasher node, now maintaining an independent set of anchors

    (a) receives a set of matching seeds from a storage node.

    (b) makes a new anchor A for each subset of seeds such that there is exactly one seed from each sequence.

    (c) extends each new anchor A by ungapped alignment.

    (d) coalesces each new anchor with pre-existing anchors.

11. Once all hasher nodes have finished receiving and building anchors, hasher nodes have to merge these anchors between them. This is the "distributed merge" step. Initially all hasher nodes contain unmerged anchors and are considered "active."

12. Active hashers are broken into "sender/receiver" pairs, such that hasher $2n$ receives anchors from hasher $2n + 1$.

13. Anchors are merged by the receiver into the pre-existing anchor set, just as new anchors were in the sequential algorithm.

14. Hashers that have finished sending are deactivated, and the remaining hashers repeat the process from step 12 until all anchors reside on a single hasher.

The final "distributed merge" step 11 above is unique to the parallel algorithm, and is the only place where additional overhead for parallelization is introduced. The memory overhead is minimal, and because the number of active hashers is halved at each iteration, the distributed merge step requires only $\lceil \log_2 N \rceil$ (where N is the number of participating hasher nodes) iterations to complete. The parallel algorithm is summarized in Figure 2.3.

Most modern workstations and servers used in cluster environments generally have a limited amount of RAM available. Therefore, Murasaki's parallelization scheme presents a useful advantage in that it allows the biggest memory requirement, the hash table, to be distributed across an arbitrary number of machines. This enables the use of proportionately larger hash tables and thereby enables fast indexing of larger sequences such as multiple whole mammalian genomes. Murasaki automatically exploits this increased available memory by incrementing the hashbits parameter (doubling the size of the hash table) each time the number of machines available in a cluster doubles.

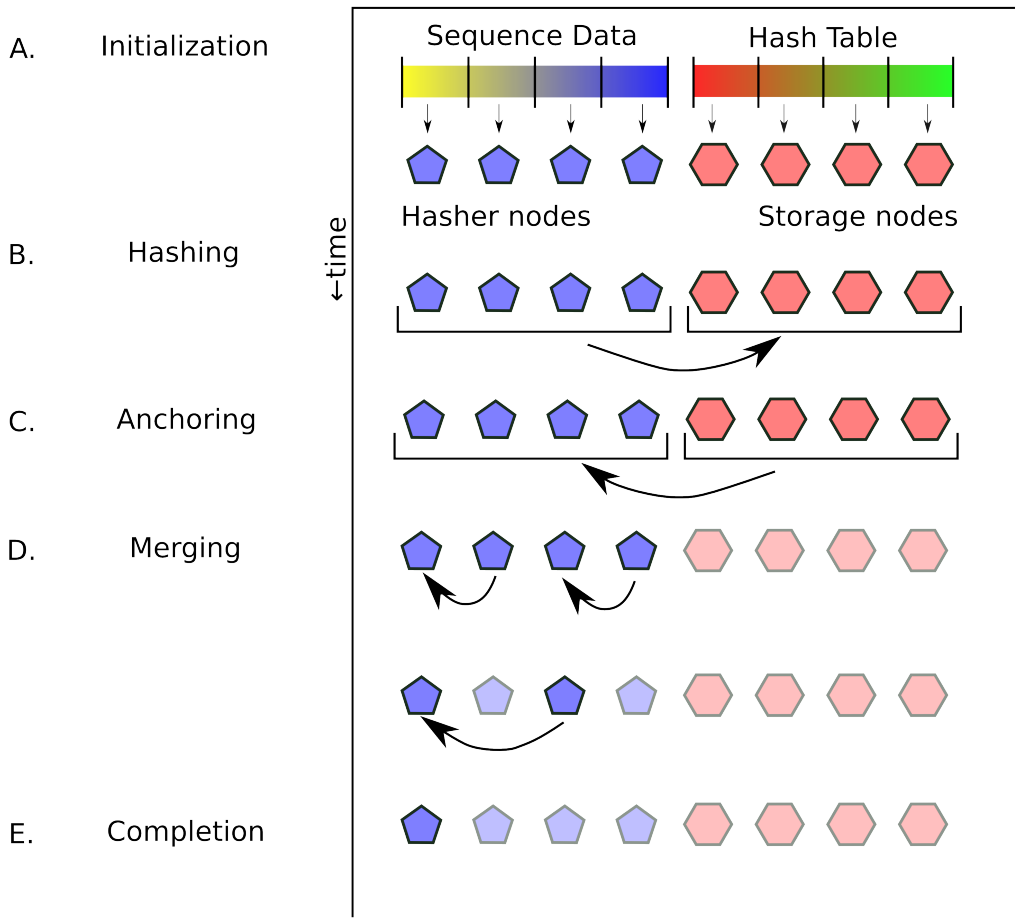

Figure 2.3: Here we show a simplified 8 node example of the parallel Murasaki algorithm. The time axis shows the progression of steps and is not drawn to scale. At **A** Each hasher node is assigned an equal part of the sequence data (depicted as yellow-blue line), and each storage node is assigned a part of the hash table (depicted as a red-blue line). At **B** nodes have been divided into "hasher nodes" (shown as blue pentagons) and "storage nodes" (shown as red hexagons). Here hasher nodes act as producers, hashing the input sequence and passing (K, V) pairs to the storage nodes which store them in the hash table. At **C**, the producer/consumer roles are switched such that storage nodes extract matching seed sets from the hash table and send them to hash nodes. Once all matching seed sets have been extracted, the storage nodes are finished and can be terminated (indicated by the lighter coloring in **D**). At this point each hasher node has an independent anchor tree. Hasher nodes are divided into pairs, with one node sending all of its anchors to the other. These anchors are merged using the normal coalescing algorithm. Once a hasher has finished sending, it can be terminated. Because the number of hasher nodes is halved at each iteration, this merge step finishes in $\lceil \log_2 N \rceil$ iterations, where N is the number of hasher nodes. At **E** only one hasher node remains which handles any additional scoring and filtering of anchors, and outputs the final result.

2.2.3 Hash function generation

As described above, the choice of hash function determines the efficiency with which the hash table can be utilized. Therefore, it is important that the hash function be chosen with care. The ultimate goal of the hash function is to provide a means of reducing the number of operations necessary to identify all seeds matching a given seed K_i. Therefore, we would like each key of the hash table to be shared by as few seeds as possible. The "ideal" hash function provides minimal collisions while requiring only minimal computation to calculate $H(K_i)$.

To describe our adaptive hash algorithm, first recall that input sequences in Murasaki are stored with two bits per base. Thus a "word" (the most primitive computational unit on which a CPU can operate) in a modern 64-bit CPU contains 32 bases, and a 32-bit word would contain 16 bases; however the algorithm itself works with any arbitrary word size W. The *spaced seed pattern* is also expressed in the same two bits per base format, and therefore consists of several words $P_0, P_1, \ldots, P_n$, where n is number of words required to express the pattern (n is therefore $\lceil \frac{l}{W} \rceil$ where l is the pattern length). An example seed and resulting words are shown in Figure 2.4 part A. Hashing by any of the above algorithms requires first that the bases ignored by the spaced seed pattern (the 0s) are masked (or eliminated). This can be accomplished for any given location v in the input sequence by simple bitwise AND operation. Because this operation will be repeated for each position in the genome, a pattern-sized buffer (which we call a *window*) of n words ($I_{0..n}$) is prepared to facilitate this calculation. $n + 1$ words from v are copied into the window and bit shifted to align v to the initial word boundary. The spaced seed masked word S_i can be computed as the simple bitwise AND of I_i and P_i. When hashing the whole input sequence, after hashing one window, the next window can be calculated by again simply bit shifting each word $I_{0..n}$ and recalculating the bitwise AND for $S_{0..n}$.

This provides a framework for running arbitrary hash algorithms on spaced seeds. However no single one of these words alone is likely to make a good hash, as the masked bases in them provide zero entropy, and because the other bases aren't expected to be conditionally independent. To maximize the entropy of the hash, it is useful to combine words from across the breadth of the pattern. Therefore our adaptive hash algorithm generator dynamically generates hash functions in terms of a set of input pairs (i, j) in which i indicates which word of the window to select, and j specifies a bit shift to apply to that word (positive and negative values indicating right and left shifts, respectively). The hash itself is computed by `XOR`ing the result of $S_i \gg j$ (or $S_i \ll -j$ if j is negative) of all (i, j) input pairs. This process is

A. Seeds as words in memory

	Sequence in word-sized blocks (w[0] and w[1])	
Input Sequence	AGCTGGGCGTGGTTGT	GTGCACCTGTAATCCC
Spaced Seed Pattern	1100000110111001	1101100100000111
Masked Seed	AG.....CG.GGT..T	GT.CA..T.....CCC

B1. A potential hash function:

```
w[0]<<3 ^ w[0] ^ w[1]<<2 ^ w[1] ^ w[1]>>6
```

B2. Hash function input sources illustrated with respect to words

Inputs	Sequence in word-sized blocks (w[0] and w[1])	
Input 1 (w[0]<<3)	AG.....CG.GGT..T	GT.CA..T.....CCC
Input 2 (w[0])	AG.....CG.GGT..T	GT.CA..T.....CCC
Input 3 (w[1]<<2)	AG.....CG.GGT..T	GT.CA..T.....CCC
Input 4 (w[1])	AG.....CG.GGT..T	GT.CA..T.....CCC
Input 5 (w[1]>>6)	AG.....CG.GGT..T	GT.CA..T.....CCC

B3. Hash function XOR operation illustrated

XOR Inputs	DNA Representation	Binary Representation
Input 1 (w[0]<<3)	....CG.GGT..T	00000000011000101011000011000000
Input 2 (w[0])	AG.....CG.GGT..T	00100000000000011000101011000011
Input 3 (w[1]<<2)	.CA..T.....CCC	00010000001100000000000010101 0000
Input 4 (w[1])	GT.CA..T.....CCC	10110001000000110000000000010101
Input 5 (w[1]>>6)	⊕ GT.CA..T..	00000000000010110001000000110000
XOR Output (Hash)	GAACCCGTAGGTCTCG	10000001010110110010101101110110

Figure 2.4: Here we show an example hash function and how it is calculated. **A** illustrates how a seed might be stored in memory on a 32-bit machine. We show 32 bases of sequence stored in two 32-bit (16 base) words. The spaced seed pattern we're using as an example here is 32 bases long, with a weight of 16. The last line of **A** shows the input sequence after being masked by the spaced seed pattern, where the masked bases have been replaced with . (periods). **B1** shows an example hash function, expressed in C terms as a series of words (`w[0]` or `w[1]`), in most cases bit shifted left (<<) or right (>>), and collectively `XOR`'d together (the ^ operator). **B2** shows the section of sequence being selected by each bit shifted `XOR` term. **B3** shows the actual `XOR` calculation that takes place with each bit shifted term expressed both as DNA bases and in binary. The highlighted regions show positions in the hash affected by the input sequence, with the color indicating the `XOR` term from which they originated. The final resulting hash is shown on the bottom line.

illustrated with a practical example in Figure 2.4.

These hash functions themselves are simple and fast to compute; however the number of possible hash functions is extreme. For any given spaced seed pattern of length l, there are $\lceil \frac{l}{W} \rceil$ choices of word, and $2(W-1)$ choices of shifts for each hash input pair, and therefore $2l(W-1)/\lceil \frac{l}{W} \rceil$ possible pairs. Because input pairs are combined by `XOR`, applying the same input twice is equivalent to not applying it at all. Therefore there are exactly $2^{2l(W-1)/\lceil \frac{l}{W} \rceil}$ or $\mathbf{O}(2^l)$ possible hash functions. The vast majority of these hash functions are undesirable as they use an excessive number of inputs or leave some parts of the hash underutilized. Therefore finding the "good" hash functions is a nontrivial problem. Our adaptive hash generator solves this problem by using a genetic algorithm to iteratively explore the space of possible hash functions. In this approach, we create a population of (initially random) hash functions, and each cycle they are evaluated for "fitness" based on their expected entropy and computation cost (method described in Section B.1.4). The highest scoring third of hash functions are randomly combined and mutated to generate new hash functions, and the lowest scoring third are eliminated. By default we start with 100 hash functions, and repeat for at least 1000 cycles or until the marginal improvement of the best hash function drops below a given threshold.

2.3 Results

2.3.1 Experiment Design

Because Murasaki focuses solely on multisequence anchor identification, it is difficult to identify a "drop-in replacement" from existing toolchains against which to compare Murasaki. Murasaki has already been used in several projects including orthologous segment mapping (Hachiya et al., 2009), and a study of *Pseudomonas aeruginosa* that revealed the occurrences of large inversions in various *P. aeruginosa* chromosomes (Mathee et al., 2008); so it is known empirically to be a useful tool. To quantitatively test accuracy and efficiency of Murasaki we evaluated Murasaki's performance under several controlled scenarios with respect to speed and accuracy. Our tests focus on either whole genomes, or when the whole genome would be cost-prohibitive, just the X chromosome for expediency. The concerns that we address in our testing include:

1. Comparison to existing methods

2. Adaptive hash algorithm performance

3. Parallelization and scalability in cluster-computing environments

4. Performance on large inputs

Lacking a perfect drop-in replacement for an existing method, we chose to work with BLASTZ (Schwartz et al., 2003) to generate pairwise anchors and TBA (Blanchette et al., 2004) to combine BLASTZ's anchors into multi-sequence anchors when needed. BLASTZ is widely used as another Swiss-army knife of homology search, and provides options to return anchors at the ungapped-alignment stage similar to Murasaki. We cannot force BLASTZ to use longer spaced seed patterns, and recognize this is not BLASTZ's intended use, but it can be made to fulfill the same basic anchor finding functions. The combination of BLASTZ with TBA is consistent with the intended use of TBA. We use blastz.v7 with options "C=3 T=4 M=100 K=6000" to run BLASTZ with pattern settings similar to Murasaki. The "C=3" parameter skips the gapped extension and chaining steps, outputting only HSPs ("high scoring pairs"), effectively anchors just like those of Murasaki. The "K=6000" score threshold was selected based on existing studies using BLASTZ on mammalian genomes (Schwartz et al., 2003). For TBA, we used tba.v12 with default parameters and TBA's `all_bz` program to run BLASTZ with the above specifications. Murasaki's parameters, primarily "–scorefilter=6000" and "–mergefilter=100", approximate the BLASTZ settings. "Mergefilter" prevents generating anchors from seeds which would incur more than the specified number of anchors, tagging these regions as "repeats". Additionally repeat masked sequences (Smit, R, and Green, 1996-2004) were obtained from the Ensembl genome database (Hubbard et al., 2002). Although the Murasaki "mergefilter" option provides some robustness against repeats, for mammalian genomes using repeat masked sequences reduces the amount of sequence that must be hashed and stored in memory by approximately one half (see Section B.2.2).

2.3.2 Comparison to existing methods

We applied both Murasaki and the BLASTZ+TBA approach described above to the X chromosomes of eight mammals: human, mouse, rat, chimp, rhesus, orangutan, dog, and cow. We compared every combinatorial choice of two species, then every choice of three species, and so on. For the final case of eight species, we repeated the test five times to account for variability in computation time. For Murasaki we used the 24 base spaced seed pattern `101111110101110111110011` to have a pattern close to the BLASTZ level of sensitivity (the method used to choose that pattern is explained in Chapter B).

First we show that Murasaki and TBA have comparable accuracy. Because Murasaki is using different spaced seeds than BLASTZ and requires that seeds match in *all* input sequences (unlike TBA where less similar matches are introduced during progressive multiple alignment), a direct comparison of individual anchors between Murasaki and TBA is not helpful as we would not expect them to find the same anchors. However, we would expect that both Murasaki and TBA should accurately anchor areas of significant similarity such as orthologous genes, and that both Murasaki and TBA would find anchors in same vicinity regardless of gene content (i.e., anchoring the same orthologous segments). We use those two ideas as the basis for our comparison.

First, we evaluated the precision and recall of Murasaki and TBA in terms of anchors found in orthologous genes. Sets of orthologous genes were used as defined in (Hachiya et al., 2009) and retrieved from the SPEED ortholog database (Vallender et al., 2006). Here "recall" and "precision" are defined in terms of "consistency" such that each anchor overlapping a known ortholog is classified as either "consistent" or "inconsistent," and anchors not overlapping any known ortholog are neither. An anchor is counted as "consistent" if and only if it overlaps each member of the orthologous gene set. Likewise an anchor is "inconsistent" if and only if it overlaps at least one member of an orthologous gene set but fails to overlap at least one of the other orthologous genes. "Recall" is then defined as the ratio of orthologous gene sets correctly detected by at least one consistent anchor compared with the total number of orthologous gene sets. "Precision" is defined as the ratio of consistent anchors to the number of anchors either consistent or inconsistent. As a combined overall score, we compute the F-score which is defined as the harmonic mean of precision (P) and recall (R):

$$F = 2(PR)/(P + R)$$

As shown in Figure 2.5, both Murasaki and TBA are similar terms of both precision and recall. For Murasaki, that recall drops off as more sequences are added; however, precision increases significantly. The same trend is visible in TBA; however, the effect is more pronounced in Murasaki where the increase in precision is far more significant, resulting in a significantly higher overall F-score, as shown in Table 2.1. The primary reason for this difference in performance characteristics is that Murasaki anchors are calculated across multiple genomes simultaneously rather than progressively, decreasing the number of erroneous matches at the cost of some sensitivity.

Second, we used the anchors produced by each algorithm to predict orthologous segments. Orthologous segments refer to an uninterrupted re-

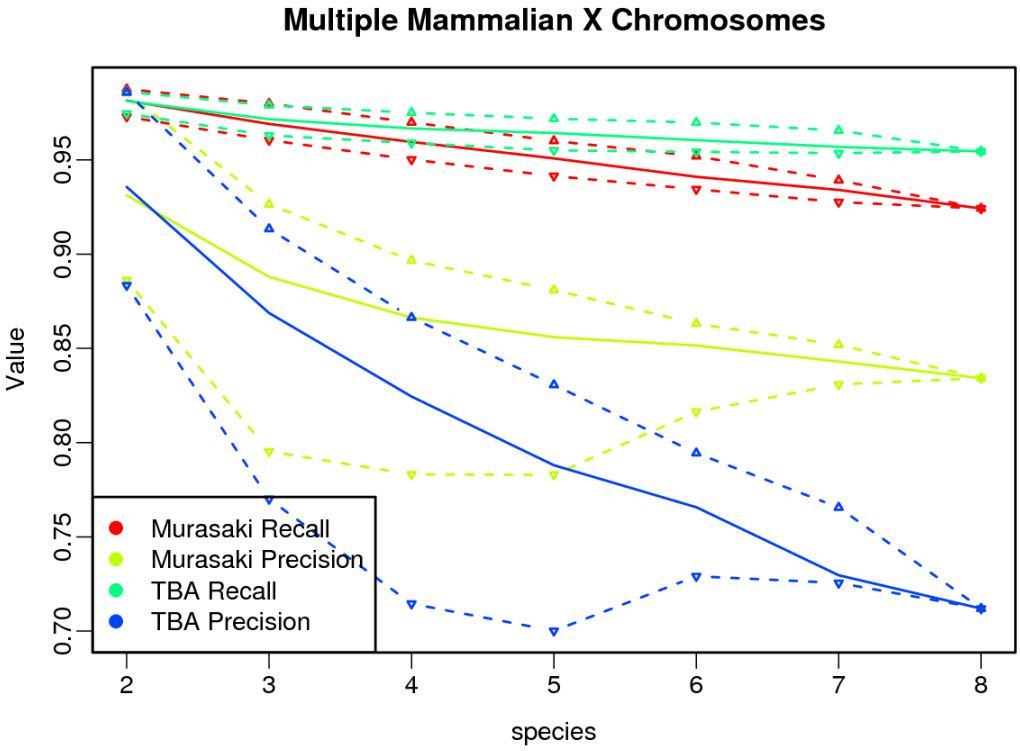

Figure 2.5: This graph examines the consistency of anchors with known orthologs when comparing varying numbers of multiple mammalian X chromosomes (from 2 to 8 different species), using both Murasaki and TBA. Consistency is evaluated in terms of recall (the percent of known orthologs anchored), and precision (the percent of anchors incident on known orthologs to correctly include the other known orthologous set members). The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles. In this graph it can be seen that as the number of species increases, both precision and recall decline with both Murasaki and TBA, however Murasaki's precision remains significantly higher than TBA.

gion of collinear homology between several genomes; that is segments unlikely to have undergone genomic rearrangement from their common ancestor(Hachiya et al., 2009). There are a number of algorithms for identifying orthologous segments, the most simple of which is GRIMM-Synteny (Pevzner and Tesler, 2003) where anchors at distances less than a user-specified threshold are merged into "syntenic blocks." In this study, we chose to use OSfinder(Hachiya et al., 2009) because it uses Markov chain models to find optimal parameters by maximizing the likelihood of the input dataset. This approach provides an anchor algorithm agnostic means to predict orthologous segments using anchors from either Murasaki or TBA.

We compared the orthologous segments from OSfinder in terms of the extent to which the resulting orthologous segments overlap as measured in base pairs, and again in terms of orthologous gene recall and precision as confirmation. As shown in Figure 2.6, the orthologous segments detected via both Murasaki and TBA share over 90% of the same bases for multiple alignments and, and over 99% at in pair-wise comparisons. To confirm that OSfinder's orthologous segments are accurate when using either algorithm, we evaluated the orthologous segments as before in terms of consistency with known orthologous genes. In terms of orthologous gene consistency, there was no significant difference between orthologous segments using Murasaki, TBA, and Roast as shown in Figure 2.7.

Finally we compare computation time. We are only concerned about time spent on anchor computation. Because in TBA we cannot separate its time spent generating progressive alignments from time spent generating multigenome anchors, we ignore the computation time from TBA and report only BLASTZ time. Consequently this slightly underreports the actual time required to generate multigenome anchors using BLASTZ and TBA, but if Murasaki is faster than the BLASTZ computation portion alone, then it is necessarily faster than BLASTZ and TBA combined; therefore, this comparison is sufficient for our purposes. The resulting computation times are shown in Figure 2.8. For pair-wise comparisons, BLASTZ is faster; however, when anchoring three or more sequences, Murasaki is significantly faster than BLASTZ. Because using TBA requires each pair-wise comparison, the computation time increases quadratically with each additional sequence under comparison. On the other hand, Murasaki's computation time increases at approximately an $N \log N$ rate (however for these cases with only two to eight mammalian genomes, only the linear N term is apparent). This is because all matching seeds are found simultaneously after being entered in the hash table together; therefore because Murasaki's runtime is bounded by the total input length N, not sequence number. The difference between Murasaki and pair-wise methods increases dramatically as the number of sequences

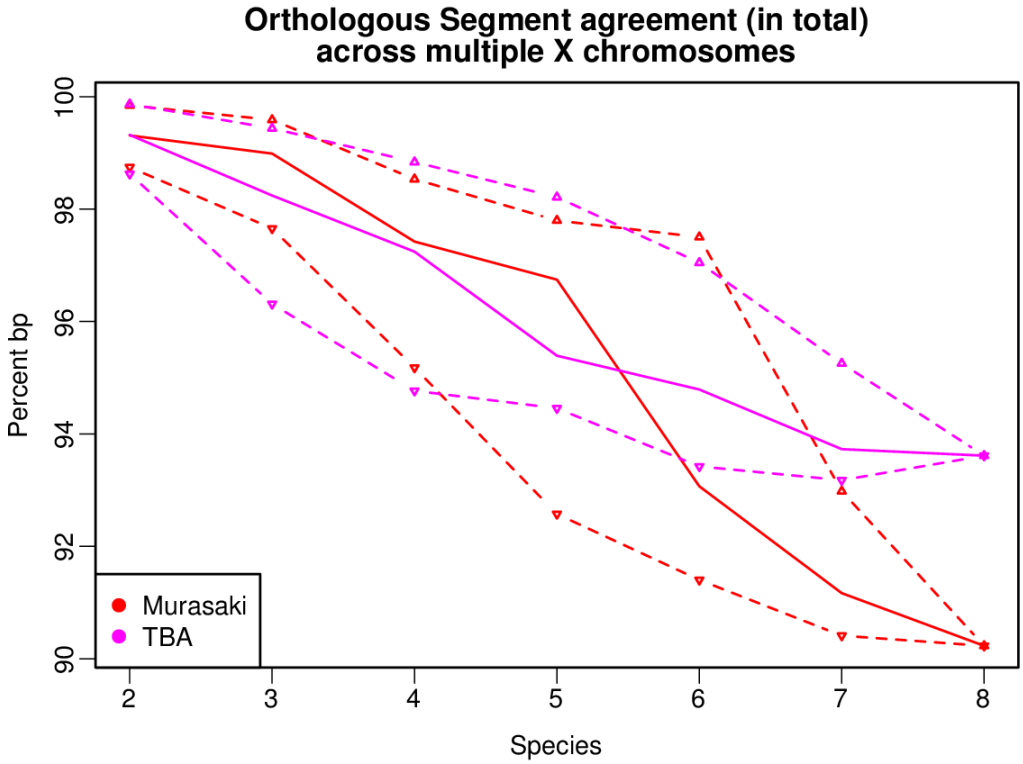

Figure 2.6: This graph shows the result of comparing othologous segments as identified by OSfinder using anchors from Murasaki and TBA. The quantities shown here are the percent of base pairs in each orthologous segment shared by the other. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles. For example, with anchors generated from all 8 species, 94% of the base pairs in the orthologous segments generated from TBA's anchors were also identified as part of an overlapping orthologous segment by anchors from Murasaki.

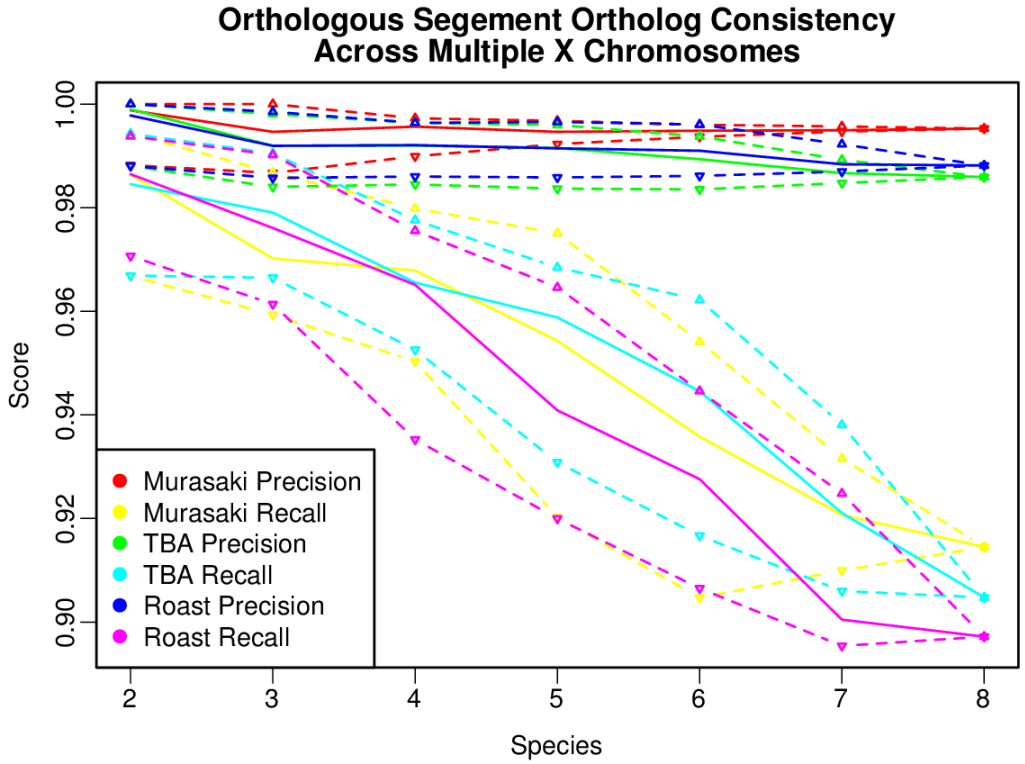

Figure 2.7: This graph shows the result of evaluating the orthologous segments returned produced by anchors from Murasaki, TBA, and Roast with OSfinder. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles. The precision and recall are calculated as described in Section 2.3.2. The orthologous segments produced using both TBA and Murasaki are nearly identical in terms of recall, however Murasaki outperforms TBA in terms of precision in our comparisons of large numbers of species. We note however that both Murasaki and TBA perform very well with all precision and recall scores above 90%.

increases. The computation times for these tests are shown in Table 2.1.

2.3.3 Adaptive hash algorithm performance

To evaluate the performance of our adaptive hash algorithm, we compared it with the standard cryptographic SHA-1 and MD5 hash algorithms, and the First-N approach. Being designed for cryptographic use, we expect SHA-1 and MD5 hash algorithms to provide near-random utilization of the key space while being more computationally expensive. To test this, we ran Murasaki on Human and Mouse X chromosomes using five different patterns, over different hashbits settings, repeating each trial four times. We then compared the number of unique keys produced by each hash function to the median number of keys produced by the adaptive hash algorithm, as shown in Table 2.2. Our adaptive hash algorithm performs within %0.05 of the cryptographic hash functions, while the naive First-N approach lags 32% behind any of the others. We find that as hash keys are used, fewer collisions require less work to invert the hash table, resulting in faster extraction times, as shown in Table 2.2. Table 2.2 also shows the computational time required to hash the input sequences under each hash function. As expected, the cryptographic functions were significantly (between 52% and 80%) slower. It is worth noting that hash computation times required by our naive First-N hash function exceeded even the cryptographic MD5 and SHA-1 hash functions. Even though calculation of our naive First-N hash function is conceptually extremely simple, it is computationally inefficient compared to hash functions that incorporate spaces using the pattern optimized approach used in the adaptive and cryptographic hash functions. The combined effect of hash time and extraction time is apparent in Table 2.2, showing the total processing time required using each hash function. Overall run-time using the adaptive hasher was 15% to 20% faster than the cryptographic hashers, and 23% to 30% faster than the naive approach. Percentages of key utilization, and times for extracting and hashing are shown relative to our adaptive hasher in supplemental Figures S1, S2, and S3.

We also tested Murasaki on Human and Mouse X chromosomes using different random patterns of lengths from 48 to 1024 at multiples of 16. Five random patterns were generated for each length, and each pattern had a weight 75% of its length. Each test was repeated three times to reduce the variability in timing. As shown in Figure 2.9, the adaptive hash functions consistently outperformed MD5 in hashing time while maintaining an extraction time almost identical to MD5. The stair-step appearance of the hash times of MD5 is due to the way that MD5 processes input in blocks, and when input lengths roll over such a block boundary, a new round of calcula-

Table 2.1: Multiple X Chromosome Test Results

	BLASTZ + TBA				Murasaki			
Species	Time (s)	Recall	Precision	F-Score	Time (s)	Recall	Precision	F-Score
2	154	0.981	0.936	0.956	349	0.981	0.931	0.954
3	459	0.972	0.869	0.916	457	0.969	0.888	0.924
4	906	0.967	0.824	0.890	587	0.960	0.866	0.910
5	1489	0.964	0.788	0.869	806	0.951	0.856	0.899
6	2276	0.961	0.766	0.852	961	0.941	0.852	0.892
7	3215	0.957	0.730	0.828	1133	0.934	0.843	0.889
8	4437	0.955	0.712	0.816	1516	0.924	0.834	0.877

This table shows the median statistics from the multiple X chromosome test. Median total computation time, recall, precision, and F-score are shown for each number of species compared using both Murasaki and BLASTZ+TBA. The BLASTZ+TBA computation time includes only the BLASTZ portion of the calculations.

Table 2.2: Total Computational Time by Hash Algorithm and Hashbits

	Hash algorithm			
Statistic measured	Adaptive	MD5	SHA-1	First-N
Hash Time (s)	124.908	188.449	208.954	218.202
Extract Time (s)	200.554	197.254	196.82	218.334
Total Time (s)	325.798	386.706	405.385	437.65
Hash keys used	1	1.00018	1.00018	0.71606

This table shows the median total computational time, along with separate times to hash and extract anchors required by different hash algorithms when anchoring human and mouse X chromosomes. The final line shows the median number of hash keys used by each hash algorithm relative to the number used by the Adaptive hash algorithm.

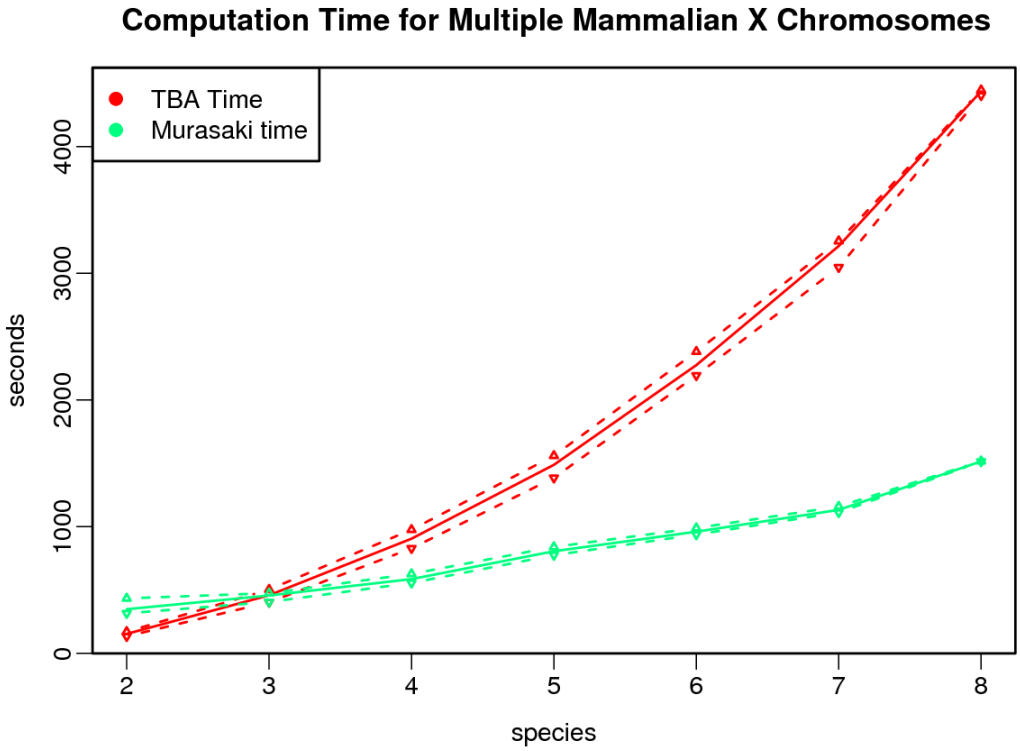

Figure 2.8: This graph compares the computational time required to compare multiple X mammalian X chromosomes using Murasaki and the BLASTZ component of TBA. Because TBA requires all pairwise comparisons of the genomes under alignment, the time required for TBA grows quadratically, while Murasaki's time is near linear. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles.

tions is incurred. In contrast the hash time for the adaptive hash algorithm grows very slowly with regard to pattern length, because estimation of the expected hash function entropy allows Murasaki to predict the point at which adding additional inputs no longer provides significant gains for the current hash key size. The percentages of keyspace used with these long patterns is shown in supplemental Figure S4.

2.3.4 Scalability in cluster-computing environments

Based on the parallel algorithm design, we expect the peak efficiency of the parallel computation to vary depending on the interconnect speed of the nodes; however because the computationally intensive tasks can be split into independent sets and divided evenly between nodes, we expect the execution time to decrease by a multiple of the number of active nodes. In other words, for p processors and a given input, where Murasaki finish in time $T(p)$, we expect the speedup $S(p)$ to grow linearly with p as in $S(p) = c \times \frac{T(1)}{T(p)}$ for some constant c.

To test Murasaki against this hypothetical performance, we used Murasaki to anchor Human and Mouse chromosomes using between 2 and 40 processors across 10 machines. We used OpenMPI on Torque as our MPI implementation, and each CPU was a dual core Opteron 2220 SE, with two CPUs per machine (ie. 4 cores per machine) and had between 16GB and 32GB of RAM available. In fact, because the amount of RAM available for use as a hash table grows with the number of machines used, the actual speed-up may be greater than linear for large inputs and large numbers of processor elements. To test the scalability of Murasaki on large inputs, we ran our tests using and the whole human and mouse genomes across the largest number of CPUs we had available.

Figure 2.10 shows the resulting decrease in wall clock time required as the number of processors increases, and the coressponding speedup value. Because the whole genome comparison requires too much memory for any single machine in our cluster, $T(1)$ is estimated to be $4 \times T(4)$. We have fitted least-squares linear regression lines to each set of values, and found the speedup constant c to be 1.000368 with $R^2 = 0.9984$. While the $T(1)$ value is available only as an estimate, the close fit to a linear model shows that the algorithm scales favorably. Critically the parallel efficiency ($E(p) = \frac{T(1)}{pT(p)} = \frac{S(p)}{p}$) shown in Figure 2.11 appears to increase with respect to the number processors, the most desirable yet elusive pattern in parallel algorithms. This increase in efficiency is due in part to the increasing hash table size; however all tests with $p \geq 10$ have access to all the machines' memory and utlize the

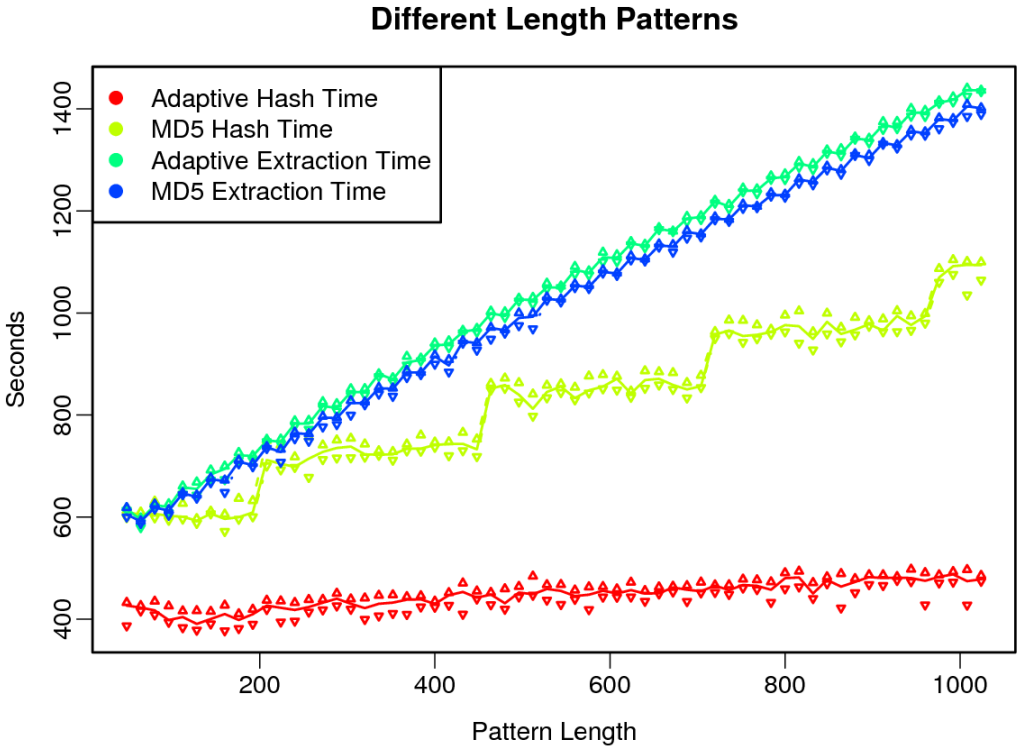

Figure 2.9: This graph shows the hash and extract computation time required to compare human and mouse X chromosomes using very long patterns, and the difference between MD5 and Adaptive hash algorithms. The difference between MD5 and Adaptive in hash time grows significantly with pattern length, whereas the difference in extraction time is minuscule compared with the overall time required.

same 2^{32} entry hash table, therefore we speculate that the remaining increase likely relates to improved scheduling and cache performance as each nodes' work becomes increasingly localized.

2.3.5 Performance on large inputs

To test the scalability of Murasaki for full multiple genome comparisons, we repeated the **comparison to existing multiple alignment methods** test on eight mammals (human, mouse, rat, chimp, rhesus, orangutan, dog, and cow), however this time using the whole genome rather than just the X chromosomes. Again, for BLASTZ and TBA we measure only the computational of BLASTZ alone. We used the same pattern and other settings as before; however, this time we ran Murasaki in parallel across 10 machines using 40 cores as in the scalability test above, using a fix hash bits setting of 29. We report the total median CPU time used by Murasaki and BLASTZ along with recall, precision, and F-score statistics in Table 2.3 for all combinations for each number of sequences. The scalability cost of the BLASTZ+TBA combination is even more striking in this case as BLASTZ is unable to compare input whole genomes, requiring the user to compare each chromosome combination (Human-1 and Chimp-1, Human-1 and Chimp-2, etc.) for each species combination (Human and Chimp, Human and Rhesus, etc.). Consequently the resulting graph of these times shown in Figure 2.12 makes Murasaki appear nearly constant by comparison to BLASTZ. When evaluated against gene orthology dataset as in the test cases above, the overall first, second, and third quartile F-Scores from all combinations of these whole genomes are 0.832, 0.861, and 0.896 respectively, leading us to believe that these anchors are approximately as accurate as those found in the X chromosome tests above.

The eight species comparison anchors (drawn using GMV(Osana, Popendorf, and Sakakibara, In preparation)) are shown in Figure 2.13. All of these comparisons are available for download and interactive browsing with GMV(Osana, Popendorf, and Sakakibara, In preparation) from the Murasaki website (`http://murasaki.dna.bio.keio.ac.jp`).

2.4 Discussion

2.4.1 Choice of comparison algorithm

Because BLASTZ is optimized for pair-wise comparisons, it can be expected to do well on a small number of inputs. However, because all-by-all com-

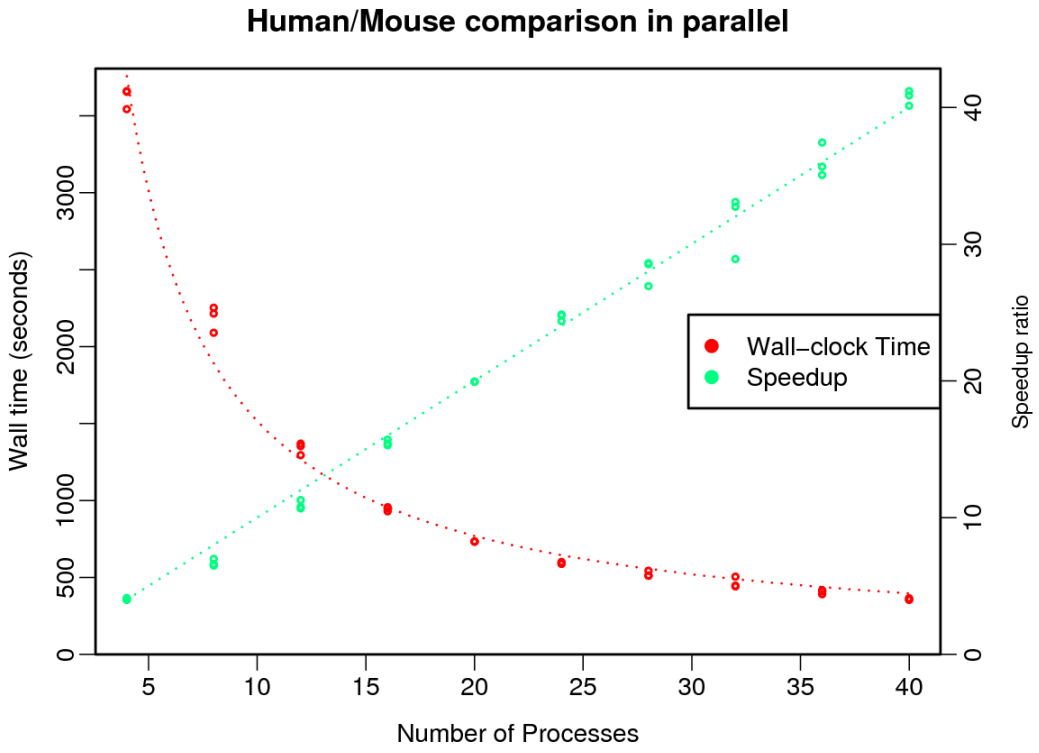

Figure 2.10: This graph shows the computational time required for a comparison of human and mouse genomes using different numbers of processors. The wall clock time is shown in red using the left axis with the corresponding "speedup" ($S(p) = c \times \frac{T(1)}{T(p)}$) shown in green using the right axis. Least-squares regression lines have been fitted to each dataset, highlighting the near perfectly linear speedup and inversely decreasing wall clock times.

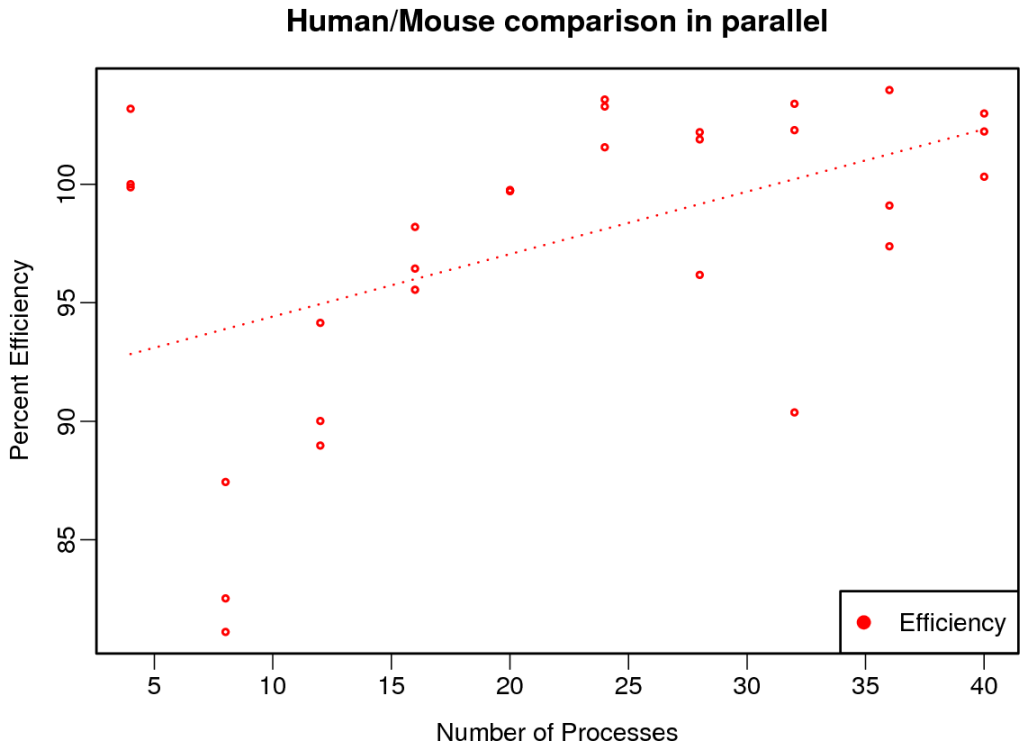

Figure 2.11: This graph shows the parallel computation "efficiency" ($E(p) = \frac{T(1)}{pT(p)} = \frac{S(p)}{p}$) achieved during comparisons of human and mouse genomes using different numbers of processors. Least-squares regression lines have been fitted to each dataset, highlighting increase in effiency with respect to number of processors.

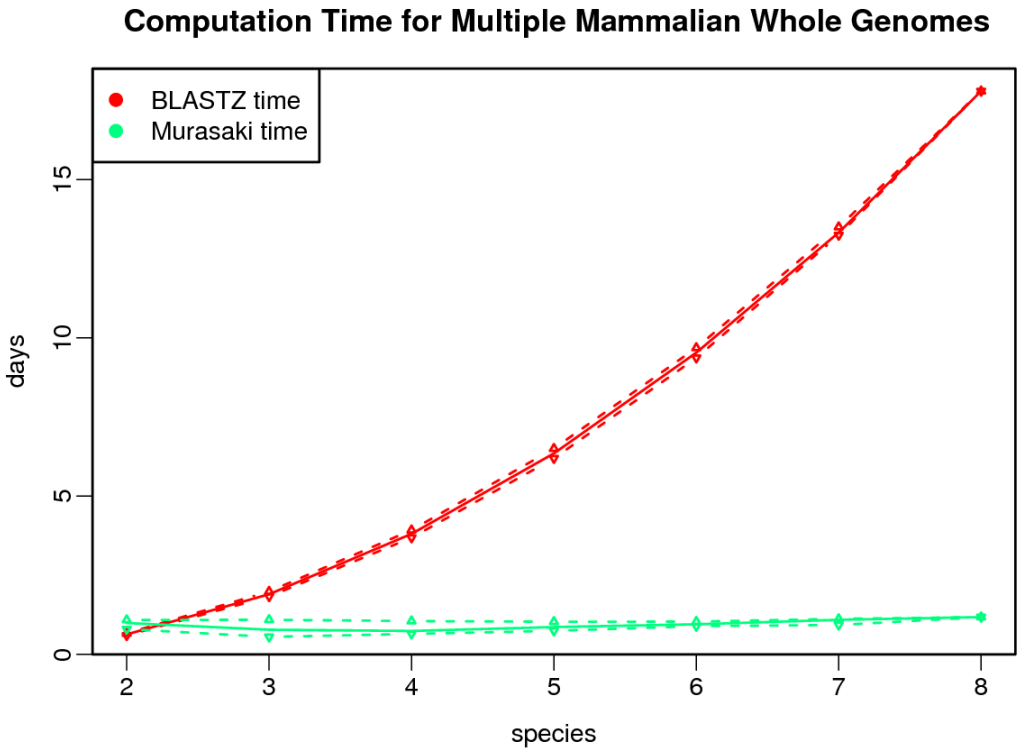

Figure 2.12: This graph shows the median CPU time in days required to anchor different numbers of mammalian whole genomes using TBA and Murasaki. The times for TBA include only the time spent on pairwise BLASTZ comparisons. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles.

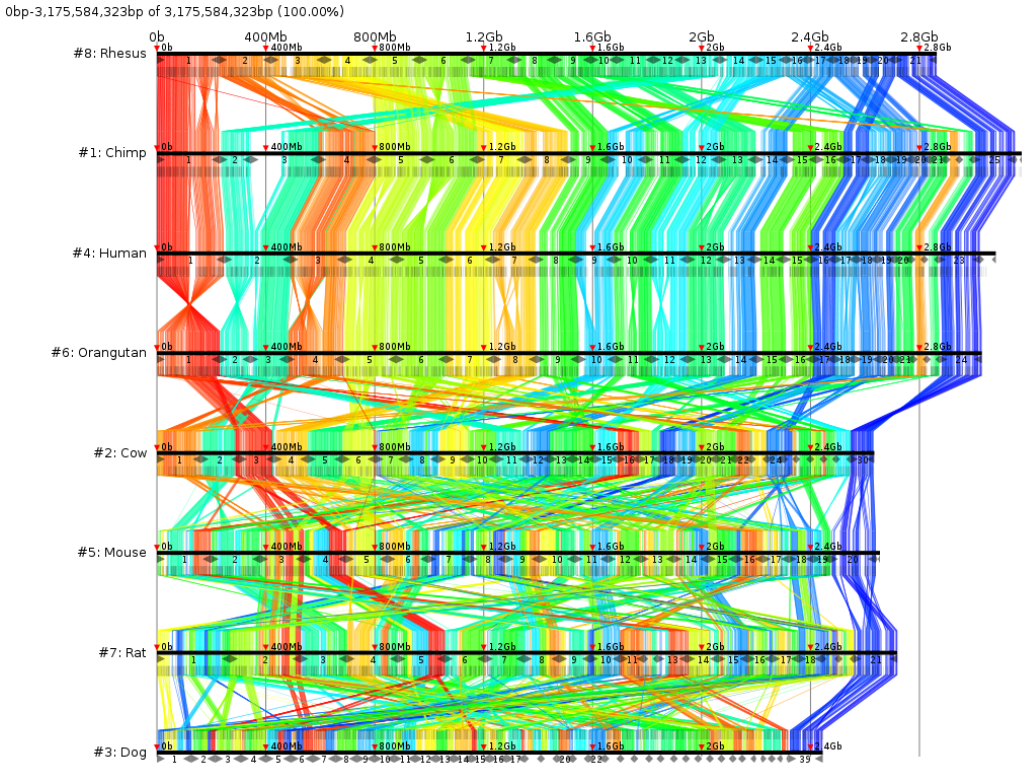

Figure 2.13: This figure shows the resulting anchors from our comparison of 8 mammalian genomes (from top to bottom): rhesus, chimp, human, orangutan, cow, mouse, rat, and dog. Anchors are drawn as colored lines from one sequence to the next. The color is determined by the anchor's position in the first (rhesus) genome, making it easier to see rearrangements and where the other genomes are related. Chromosomes are denoted by the number shown between ▶ and ◀ symbols along each genome. The sex chromosomes are shown at the right end (e.g., 23 (X) and 24 (Y) for human). For a larger image see `doi:10.1371/journal.pone.0012651`.

parisons are required to generate multiple alignments, the time required is expected to grow quadratically with the number of input sequences. In contrast, Murasaki is designed to compare an arbitrarily large number of genomes simultaneously, and assuming a linearly bounded number of anchors, the computation time for Murasaki is expected to be approximately $O(NlogN)$ for a total input length of N.

It might then seem that rather than BLASTZ, a better comparison of Murasaki would be to a natively "multiple" alignment program like Mauve; however it is important to note that Murasaki performs a fundamentally different function than Mauve in that Mauve aligns whole collinear regions bounded by *unique* anchors. While these anchors are in some ways analogous to Murasaki's anchors, the requirement of "unique anchors" puts Mauve in a fundamentally different arena, where its strength lies in alignment rather than anchoring. Also, while Mauve is well suited to bacterial genomes, it is not well suited for mammalian scale genomes (it is reportedly not impossible (Darling et al., 2004), but this use is not recommended, it does not work without applying some undocumented options to perform the necessary out of core sort, and we could not replicate or verify the results).

We also tested another alternative from the TBA package called Roast which appears to implement the method described in (Miller et al., 2007) which builds a multiple alignment based on pairwise comparisons between a reference sequence and all other sequences, thus in theory requiring time linear with respect to the number of sequences, similar to Murasaki. However due to an apparent bug in the implementation, Roast is actually worse than TBA in some cases. Even assuming that the were fixed, however, the fragmentation required to compare sequences via BLASTZ results in time requirements which grow several times faster than Murasaki at whole genome scales. The results from our fixed version of Roast and the native Roast comparison are included and discussed in Section B.2.3.

2.4.2 Bottlenecks in parallelization

Under the parallel algorithm, when hasher nodes send seeds to storage nodes, the choice of storage node is determined by the hash key. This means that balance and contention between storage nodes is, ultimately, determined by the input sequences. For example a sequence containing only one type of base (e.g., 4 Gbp of AAAAA) would necessarily all get sent to the same storage node, causing a less than optimal distribution of storage and heavy contention for that node. This is in fact the worst case, and highly improbable with real-world genomes, but similar factors can unbalance the load between storage nodes. This problem is mitigated by the near uniform random output

of Murasaki's hash functions, making it approximately equally unlikely that any two given seeds share the same node, but it does not help the worst case. A modification to the hash table data structure might allow storage nodes to dynamically update their active hash table region, and redirect overexpressed seeds to less heavily loaded storage nodes. This would of course require some additional overhead.

2.4.3 Parallel overhead

While adding machines to a cluster can increase the amount of available RAM indefinitely, the storage of the input sequences themselves in memory incurs a constant cost per machine added. Shared memory is used to mitigate this cost by loading only one copy per machine (rather than per processor), and input sequences are stored 2 bits per bp. However, as the size of the input sequence grows, at some point merely loading all the sequences into memory exhausts the system's memory. Thus the smallest memory machine in the network effectively limits the maximum input size of Murasaki. For example loading all 3.1×10^9 bp of the human genome takes about 738MB. Comparing ten mammals requires at least 7GB per machine, and that is not including any space for the hash table.

2.5 Conclusions

We have shown that our anchoring algorithm Murasaki produces accurate anchors across multiple genomes with a computational efficiency significantly greater than existing methods. Its adaptive hash function generation algorithm provides an efficient method to use arbitrary spaced seeds of any length with collision rates close to pseudorandom one-way cryptographic hash algorithms at a fraction of the computational cost. Additionally, our method is highly scalable, allowing whole computer clusters to be fully utilized for large-scale multiple genome comparison.

Table 2.3: Mammal Whole Genome Comparisons

Species	TBA-BLASTZ CPU Time (days)	Murasaki CPU Time (days)	Recall	Precision	F-Score
2	0.632	0.991	0.971	0.910	0.925
3	1.901	0.780	0.953	0.863	0.900
4	3.808	0.741	0.922	0.832	0.867
5	6.351	0.864	0.887	0.807	0.840
6	9.534	0.951	0.855	0.790	0.818
7	13.328	1.093	0.824	0.768	0.797
8	17.796	1.180	0.790	0.764	0.777

This table shows median computation times and accuracy for mammal whole genome comparisons with respect to each number of species under comparison. Recall, precision, and F-Score were calculated from Murasaki anchors only.

Chapter 3

Samscope: An OpenGL based real-time interactive scale-free SAM viewer

3.1 Introduction

Next generation sequencing workflows often involve mapping reads onto reference genomes using tools such as SHRiMP2 (David et al., 2011), Bowtie (Langmead et al., 2009), and others. Mapping determines the likely point of origin (or origins) of a given read. Despite the multitude of different mapping methods and software, the SAM (*Sequence Alignment/Map*) format (Li et al., 2009) and the associated binary encoding (BAM) have emerged as the *lingua franca* of next generation sequencing (NGS) mapping file formats. For many projects using SAM data, it's desirable to visually inspect the results of mapping for quality control and exploration. However, because a single NGS run can provide millions of reads from billions of bases of genome sequence, simply opening up a SAM file and making sense of the content is a nontrivial problem. The two main problems in visualization of a SAM data set are:

1. Sam files are structured in terms of reads. To calculate coverage of a given base, we have to look at *each read* and see which bases it maps to, then count how many times that base has been mapped.

2. Most computers only have 1000 to 2000 pixel wide displays with which to visualize billions of data points.

As visualization is a common need, a variety of tools have been introduced to view SAM data in different ways. For example "samtools tview" (Li et al.,

2009) provides an interactive text-based viewer which shows each base of each read and reference genome as a character in a text terminal. This can be useful for inspecting narrow regions (about 100 bases) with fewer reads than terminal rows (about 30), but not helpful for examining larger regions or deeper coverage. "Tablet" (Milne et al., 2010) and its close cousin "IGB" (Nicol et al., 2009) both provide Java based graphical interfaces for drawing reads against a reference sequence where each base is represented as a colored rectangle. Both can summarize overall coverage with a secondary visualization track. However both Tablet and IGB draw each read similar to "samtools tview", limiting their speed and effectiveness when a large number of reads would be in view. "Integrative Genomics Viewer" (IGV) (Robinson et al., 2011) can load SAM/BAM files providing detailed inspection capabilities, and provides a "mean coverage" track given proper pre-processing, which can scale to arbitrary genome sizes. For applications like ChIP-Seq (Pepke, Wold, and Mortazavi, 2009) or RNA-Seq however, "*mean* coverage" is not necessarily helpful and hard to use at large scales, as most coverage values are at zero or near zero.

We needed a flexible method to visualize and interactively inspect various features from large numbers of reads across mammalian-scale genomes while addressing the problems above, so we developed our own approach in Samscope.

3D computer graphics addresses a similar problem when drawing textures on distant 3D objects: how to efficiently generate a reasonable approximation of millions of points of color data into one screen pixel. A solution known as MIP mapping (Williams, 1983) has become a mainstay of modern 3D rendering; in it a series of filtered copies of each texture are pre-calculated at exponentially decreasing resolutions. Thus when a distant object is rendered, rather than sampling millions of points to calculate the combined contribution to one screen pixel, an approximation is achieved with just a few samples from a lower resolution copy. We apply the MIP map concept to genome visualization in Samscope.

3.2 Methods and Implementation

Samscope adopts a layer-based display model, where each layer reflects a SAM mapping feature, such as coverage. Layers are stored as BAM MIP Maps ("BIPs") on disk in a binary format allowing instantaneous navigation with minimal memory requirements. Multiple layers can be displayed simultaneously as different colors, and in multiple synchronized windows. This layer-based design makes it simple to display results from multiple SAM files

as different layers, and visually compare results from different experiments. Samscope supports a variety of feature calculations for different applications. For example, in ChIP-Seq the difference in number of reads mapped to the forward strand compared to the reverse strand (what we term "polarity") results in a characteristic zero-crossing inverted-peak pair which, in conjunction with coverage, often reflects protein binding sites (Pepke, Wold, and Mortazavi, 2009). Samscope allows easy composition of data layers into compound data layers, such as polarity; "forward" and "reverse" count layers are generated, from which overall "coverage" (the sum) and "polarity" (the difference) are derived (shown in Figure 3.1). This design allows for fast and memory efficient pre-processing of huge data sets, as data primitives can be stored on disk and accessed at random as needed.

To generate BIPs we adapt the traditional graphics MIP map approach to sampling one-dimensional data series. The pre-processing algorithm is as follows:

1. Calculate the full resolution series with one value for each base of reference genome, forming a series of columns with one value each.
2. Merge data from adjacent columns generating half the number of composite columns. To merge two columns, combine values from each column as value/frequency tuples. For example, if the value "3" occurred twice in each column, a single tuple {3,4} would be retained.
3. If the number of value/frequency tuples in the current column exceeds a user adjustable maximum, retain a subset; the most frequent values are prioritized, along with the extreme maximum and minimum values.
4. The merged column data and an index reference for fast random lookup are stored to disk.
5. Steps 2-4 are repeated until only one column remains.

To render a given region of data, a set of columns is selected to match the pixel width of the display window. For example, if rendering a 50Mbp region in a 1024 pixel wide window, data from the 16th "column merge" iteration is selected, such that each column represents a sample from the underlying $2^{16} = 65{,}536$ bases. Thus the display can be rendered with data from 763 precomputed columns.

This approach has some practical benefits:

- To draw one screen of P pixels, at most $O(P)$ values must be read and drawn. Fast rendering allows us to abandon scroll bars, and adopt an intuitive mouse-based "pan and zoom" interface familiar to users of Google Maps (`http://maps.google.com`).

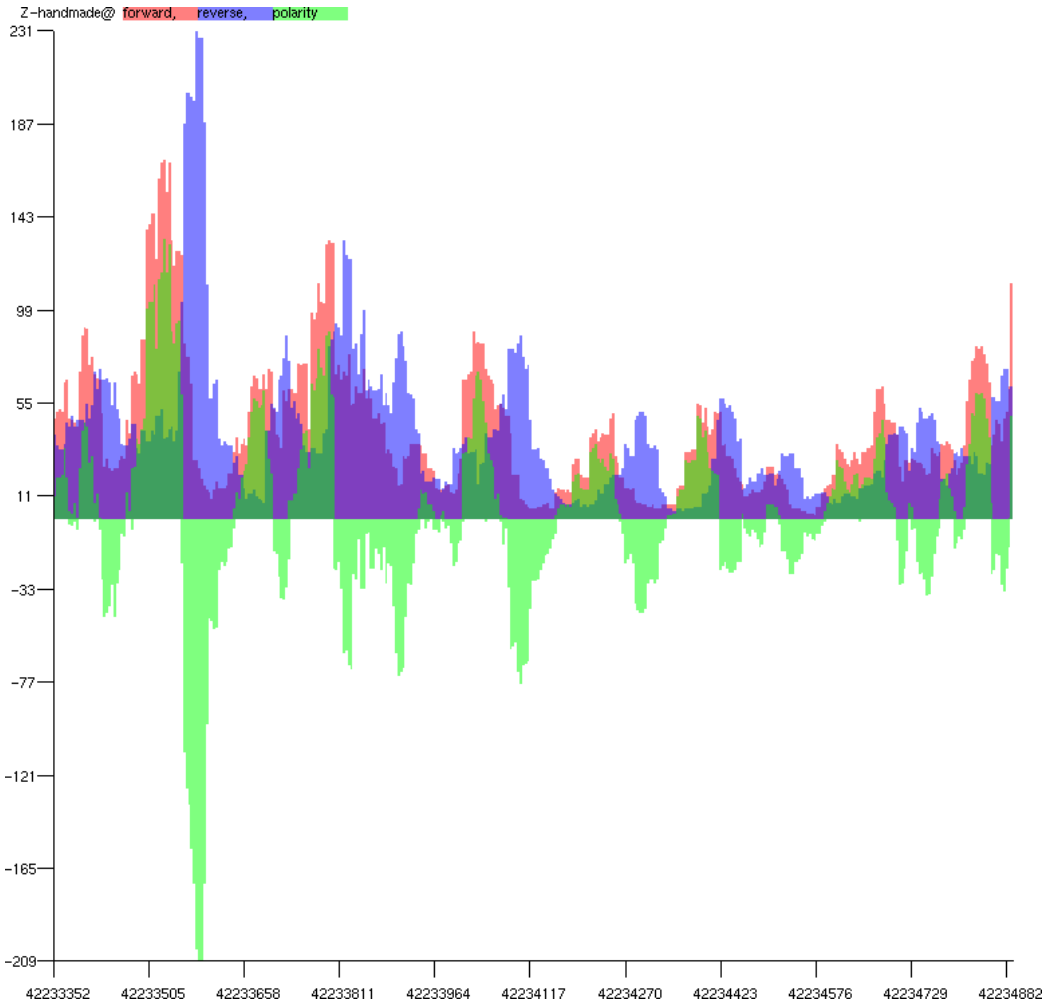

Figure 3.1: This screenshot shows "forward, reverse, and polarity" layers from a ChIP-Seq analysis of CENPA focusing on a 500bp region at in Z chromosome centromere of chicken (Shang et al., 2010). The periodic waves visible here are presumably related to structural influence of exposed binding sites.

- The pre-processed data retains multiple values per column, allows Samscope to change the rendering style at run-time to a representative spectrum of values at each column, or an average value, or maxima and minima or other arbitrary effects.
- Complex features (such as peaks in ChIP-Seq and RNA-Seq) are visible at genome-scale, and not obscured by techniques like averaging as used in previous tools like IGV and MAQ (Li, Ruan, and Durbin, 2008).
- Because only the values on the screen at any given moment need to be retained in memory, there is little need for large amounts of memory or a fast CPU.
- Just as BIP display only needs a few columns in memory at a time, BIP generation only needs the values for each pair of columns. Completed columns can be immoderately swapped to disk and will be loaded at most once more duration the generation phase. Thus Samscope can run perfectly well on an average laptop computer for both BIP generation as well as viewing.
- Finally, because empty columns do not need to store any data values, and because BIP files are stored as "sparse files" on disk, BIP files are well suited for sparse data like exomes or ChIP-Seq.

Sequence parameters (such as chromosome name and length) are obtained from the source SAM/BAM file itself, eliminating any additional sequence setup steps as are required in programs like IGV. Annotation data from GFF/GTF files are displayed if available (see Figure 3.2).

3.2.1 Displaying individual reads

While displaying aggregate statistics of mapping results is extremely useful and often sufficient for visualization, very often a use will want to examine the reads present behind an unusual or suspicious feature. As described in Section 3.1, due to the scale of read data, it's impractical and not useful to draw a full genome of reads, and features with thousands pose the same problem. The first problem, that of a long genome, is partially addressed by the existing BAM index structure from SAMtools (Li et al., 2009). The BAM index allows us to load reads that occur after a given point in a given chromosome. However, because read length (and insert length for paired end reads even more so) varies, to draw the left edge of the screen in a consistent fashion some bounded area around the current viewing window is polled. When a read is in view, its data is parsed to determine the aligned bases and calculate where mutations occurred. Per base statistics like mutation counts

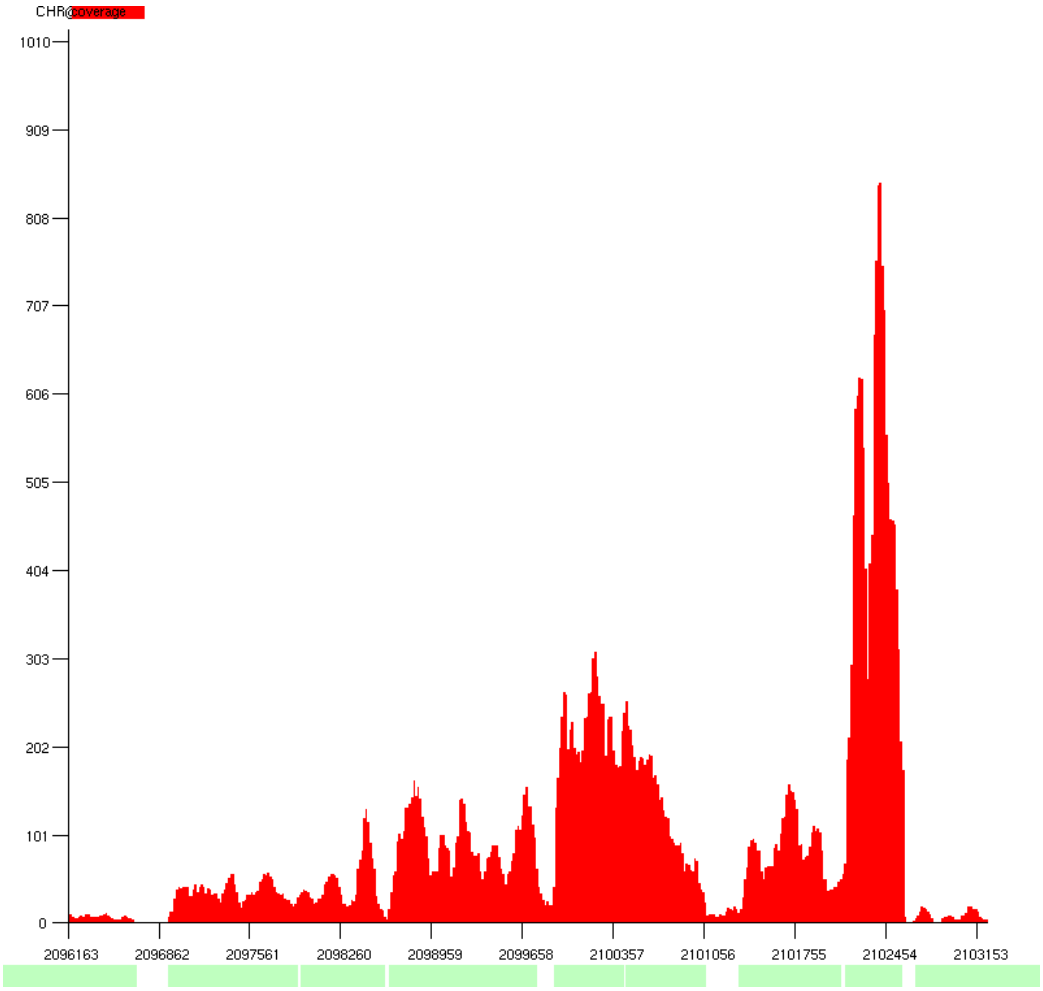

Figure 3.2: This screenshot shows coverage from an RNA-Seq experiment on a 7072bp region of *Bacillus subtilis* subsp. natto BEST195. Gene annotations are shown as green bars below the X axis.

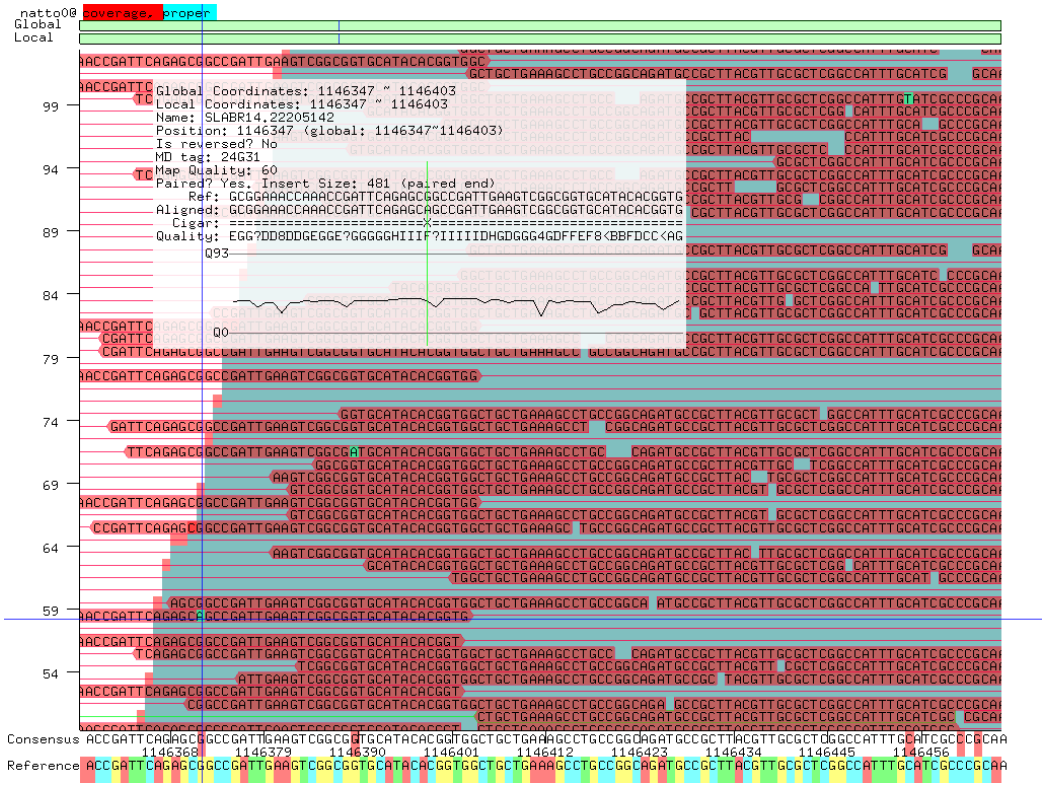

Figure 3.3: This screenshot shows a fulled zoomed in view of part of a Bacillus Subtilis variant strain of natto. The cross-hairs are moused over a variant base in one of the reads, showing that base's position in the read quality graph as well as statistics for that base. The reference sequence is derived from the SAM data and shown at the bottom of the screen.

and variant consensus calls are also calculate when per base data is drawn (or would be otherwise visible, as by mousing-over the reference/consensus bars along the bottom of the screen). This derived per-base data is cached in a Samscope specific while the read is in view. This cache allows us to address the second problem of genomic features with thousands more reads than are visible on the screen at any one time. Only reads which are physically visible (i.e. aren't obscured by attempting to draw more than 2 reads in the same pixel) are drawn and thus only a fraction of reads need be cached in memory. If the user zooms in or pans to make other reads visible, their data is fetched as necessary. Furthermore read processing is done between frame updates, so viewing frame-rate is minimally impacted. An example of the individual read rendering is shown in Figure 3.3.

3.2.2 Memory requirements

When viewing BIP data files, all necessary data is paged from disk using the POSIX `mmap` function, at most few megabytes of memory are necessary for rendering any given view of the data. If the system runs low on memory and experiences memory pressure, pages of BIP files are easily dropped from memory by the host operating system because they are loaded as "read-only" and never require rewriting to disk (as a typical virtual memory "swap space" would require). Most operating systems drop "least recently used" pages first, so most users will not even notice this occurring.

For generating BIP data files, if the input data is sorted, the data for one base of the reference sequence has been completed, that base will never be visited again, therefore it can be safely flushed to disk and dropped from memory. Because data is written using POSIX's `mmap` function, this happens automatically if the system needs more memory. If the input data is not sorted, the RAM costs are bounded by the size of the reference sequence. In general the cost is expected to be between 1 and 8 bytes of RAM per base depends on the number of layers being simultaneously generated, and the precision of the data type used for storing the layer data. 2 bytes precision per base is the default, and generating 2 layers simultaneously is typical (forward and reverse), thus an estimate of 12gb of RAM for human reference is reasonable.

3.2.3 Time requirements

The time to generate BIP data files for Samscope is effectively $O((N + M)(L))$, where N is the size of input data (in bases), M is the size of the reference genome, and L is the number of layers being generated. All bases are read in and statistics are counted for each base requiring $O(N)$ time. This works even with insufficient RAM if reads are sorted. If $\mathit{ValuePrecision} \times M$ memory is available, it completes in $O(N)$ time regardless of whether or not reads are sorted. Each MIP map contains at half the columns of the previous iteration, thus finishing in $O(M)$ time. Finally, while some layers can be generated concurrently and don't require additional passes over the $O(N)$ read data, some layers depend on prior layers (e.g. *polarity = forward - reverse*) and some layer types are not presently generated concurrently. Furthermore some users may wish to limit concurrency (see *–maxconcur*) to limit memory consumption, requiring an additional $O(L)$ passes over the data.

3.3 Results

3.3.1 BIP generation benchmarks

We've included some benchmarks on a variety of data sets (see Table 3.1) on a series of different computer environments described in Table 3.2, Table 3.3, and Table 3.4. As a practical benchmark on human reference, a coverage BIP file can be generated from 54M 120bp mapped reads on a server with 96GB of RAM in 12 minutes, or on an ordinary desktop with 6GB of RAM in 19 minutes.

Table 3.1: **Datasets**

dataset	**reads**	**read-length**	**approx. bases**	**reference size**	**prep. type**	**reference**	**notes**
Yan-11	∼54M	120bp	6.48Gbp	3.1Gbp	Exome capture	doi:10.-1038/-ng.788	There are 18 different biological replicates for this data. We show the means from runs on all of these.
Natto	24.3M	80bp	1.9Gbp	4.09Mbp	RNA-Seq	doi:10.1186/-1471-2164-11-243	
Chicken	689.9K	36bp	24.8Mbp	74.6Mbp	ChIP-Seq	doi:10.1101/-gr.106245-.110	selected reads mapped to one custom-built Z chromosome assembly only.

Table 3.2: **Intel Xeon E5540 @ 2.53GHz and 96GB RAM**
This is a Sun Fire X2270 from 2009. Not too fast, but with lots of RAM. This is running Debian Lenny.

dataset	**layers**	**time (s)**	**peak memory (GB)**	**space on disk**	**notes**
Yan-11	1	748.0	13.8	14.7GB	Coverage only.
Yan-11	9	1954.6	57.5	26.7GB	layers are all bases counts, called, minor allele freq/millis.
Chicken	4	58.0	1.1	331MB	layers are forward/reverse, coverage, polarity.
Natto	12	248	0.8	389MB	layers are bases counts, minor/major allele frequency, multimap counts, coverage, forward/reverse.
Natto	1	30.0	0 (10.0MB)	65MB	coverage only.

Table 3.3: **Dual-core AMD opteron 2220 SE @ 2.8GHz with 23GB of RAM.** This is an old IBM System x3455 from 2007. Running Debian Lenny.

dataset	**layers**	**time (s)**	**peak memory (GB)**	**space on disk**	**notes**
Yan-11	1	748.0	13.8	14.7GB	
Yan-11	9	14315.2 (∼4.2 h)	24.9	25.6GB	This pushes beyond the physical RAM capacity of this machine and demonstrates Samscope using paging in the worst possible case (as minor alleles layers depend on base data layers being calculated first).
Natto	1	35.0	0 (8.5MB)	65MB	
Chicken	4	60	2.1	331MB	

Table 3.4: **Intel Core i7 930 @ 2.8 GHz with 6GB of RAM**
This is a standard HP Pavilion desktop from 2010 with the factory standard 6GB of RAM. This computer doesn't have nearly enough RAM to hold all the active data in RAM, so we use it to demonstrate the disk-backed BIP generation capabilities of Samscope. This is running Debian Squeeze.

dataset	**layers**	**time (s)**	**peak memory (GB)**	**space on disk**	**notes**
Yan-11	1	1145.3	5.7	14.7GB	This demonstrates paging in less extreme conditions (only 1 layer to page).
Yan-11	9	17853.0	5.7	25.6GB	Not significantly worse than the 23GB machine above.
Natto	12	218.1	0.9	389MB	
Natto	1	27.0	0 (21.5MB)	65MB	
Chicken	4	58.0	1.8	331MB	

Environment

All disk access is to a central NFS server (backed with ZFS running lzjb compression) over gigabit ethernet. These trials used Samscope 1.4.6.1.

3.4 Discussion

Samscope's BIP format is similar in its goal to the BigBed and BigWig (Kent et al., 2010). However where BigBed/BigWig use B-trees and a complex data layout suitable for remote access over HTTP, Samscope uses a much simpler flat file format designed for local access through the POSIX **mmap** function. This design allows the entire process to run efficiently on commodity hardware. Samscope can be more efficient for its purpose because where BigBed is designed to contain any arbitrary annotation (e.g. scores from overlapping alignments, etc.), Samscope focuses on per base aggregate statistics (e.g. coverage, polarity, etc.). We find the BigBed and BIP formats to be complementary in this sense.

Chapter 4

Conclusions and Future Work

In this dissertation we've examined two different challenges that arise when working with large scale genomes: the task of comparing the genomes themselves, and the task of making large genome scale data understandable to human users.

4.1 Multiple genome comparison

As the cost of sequencing continues to fall, the number of complex genomes such as plants and mammals sequenced is expected to continue increasing exponentially. Given that processor clock speeds are no longer increasing, in order to keep pace with the growing number of new sequences the bioinformatics community must consider alternatives that can scale with distributed computing solutions. In Chapter 2, we demonstrated a highly scalable approach for strong homology search across multiple genomes. We showed that its accuracy for tasks like orthologous segment identification is comparable to existing methods, while being far more efficient in terms of resource utilization. We showed that performance scales linearly when run on concurrently on a cluster network, which is the nearly optimal best case outcome for distributed computing designs.

4.1.1 Future work for multiple genome comparison

Selection of an appropriate seed pattern remains a challenging problem for applying Murasaki and similar matching algorithms. Selecting a longer pattern will likely lead to lower recall of distant homologies, while selecting a shorter pattern can lead to reduced precision and slower performance as unrelated regions are considered as potential anchors. Hash table based al-

gorithms in the past have tried to avoid this problem with techniques like "Daughter Seeds" (Csuros and Ma, 2007) where multiple hashes are calculated and multiple entries stored for each location of a genome. Unfortunately this naturally carries the significant cost of extra hashing and hash table storage for limited gains in sensitivity. Recent advances in algorithms supporting suffix arrays as practical large scale text indexes (Ferragina and Manzini, 2000) have spurred development of BWT (Burrows-Wheeler Transform) (Burrows and Wheeler, 1994) based alignment algorithms including LAST (Kiełbasa et al., 2011) which exploit BWT to create compressed indexes which require a fraction of the memory ordinarily required by hash tables. LAST also uses these suffix arrays to avoid the question of selecting a single seed by exploring multiple matching suffixes. This BWT-based index approach works very well for pairwise alignment, and creating multiple indexes (or including all sequences in one index) would seem to provide a straight forward approach to multiple alignment. Suffix array creation is a relatively expensive task, however recently a new parallel approach for construction of suffix arrays has been introduced (Kulla and Sanders, 2007), which suggests a potential way to help solve the index creation problem. This, combined with a *q-gram lemma* based matching approach, like that used in SHRiMP2 (David et al., 2011) where k matching *q-grams* inside a given length of sequence indicate the presence of similar subsequences, could also be used to provide more sensitive search for alignments including insertions and deletions (indels). The parallelization of the search for nearby *q-mers* may however prove to be a challenging bottleneck to large scale homology search. The Murasaki algorithm as is remains extremely fast and efficient for cases where single pattern based seed matching is appropriate (such as relatively closely related sequences or searches for strongly conserved regions), however implementation of a compressed suffix array index on top of the Murasaki parallel anchor building algorithm could prove to be very powerful for remote homology search.

4.2 Visualization of large genomic data sets

In Chapter 3 we introduced a new approach to visualizing genome next generation sequencing data. Our approach introduces a novel data format and browser which allows extremely fast drawing of data sets to the screen. This allows researches a fluid intuitive browsing experience where they can explore their data visually, easily notice patterns, and then investigate in more detail. Because data retrieval and display can run quickly using our data format, users can easily display multiple data sets simultaneously and compare fea-

tures visually all on a commodity PC. The interface enabled by this approach is extremely intuitive, as evidenced by a similar a design that appeared after our publication in MGAviewer (Zhu et al., 2012).

4.2.1 Future work for visualization of large genomic data sets

While Samscope is fast and highly efficient once preprocessing is done, for large genomes with lots of data, preprocessing can take a few hours and use dozens of gigabytes of disk storage. While this is reasonable for most investigators of a particular problem, users wishing to share their Samscope view with the world must also distribute these large data files, which can be time consuming or impractical. BigWig and BigBed (Kent et al., 2010) provide an R tree based format designed for distributing only the data currently being viewed over an internet connection. This format could be modified to include Samscope-style arbitrary data sets, however the caching memory and architecture development necessary to create a Samscope-style viewer instantly responsive to mouse movements would likely be considerable, and increase the overall minimum running requirements. Rather, we propose that the the development of a sparse index format (in place of the existing dense index format which requires entries for all columns whether populated or not) would allow the efficient distribution of small subsets of BIP files. Furthermore, if fashioned as something like a B^*-Tree (Knuth, 1973), the index could be made efficient for offline remote storage, similar to BigBed. The cost for generating such an index might, however, be significantly higher. The existing Samscope BIP format is extremely efficient systems with fast access to local storage (even over NFS), and the local browsing enabled by this format provides a complementary format to BigBed.

Acknowledgements

I'd like to acknowledge the help and support of several people who made this dissertation possible. Yasubumi Sakakibara has been helpful to me from the very beginning, welcoming me to his lab and helping me navigate the various avenues of Japanese bureaucracy. He has helped me in in my research as well as in my personal life. I'd also like to thank Yasunori Osana who worked with me on many research projects and introduced me to many fundamentals of computer architecture and engineering. Yasunori Osana has also taught me much about Japan in general and has been a great personal support. And finally I'd like to thank Tsuyoshi Hacchiya, who has been welcoming to me from my beginning in Sakakibara Lab, and has always been helpful in research and as a personal friend. Thank you all.

Appendix A

Abbreviations

CPU	central processing unit.
FPR	false positive rate.
GPU	graphics processing unit.
IGV	"Integrative Genomics Viewer" (Robinson et al., 2011).
IGB	"Integrated Genome Browser" (Nicol et al., 2009)
MIMD	multiple instruction, multiple data (c.f. SIMD).
MIP	Latin phrase *"multum in parvo,"* meaning "much in little."
NGS	next generation sequencing/sequencer
SAM	"Sequence Alignment/Map" a file format designed for storing mapped NGS read data.
SIMD	single instruction, multiple data (c.f. MIMD).
TPR	true positive rate.

Appendix B

Murasaki: Supplemental details

B.1 Implementation

B.1.1 Hash Functions

Because the difference between the naive First-n approach and the others dwarfs the difference between any of the other differences, we look at only the 3 high-performing algorithms in Figure B.1. We note that our adaptive hash algorithm starts to diverge only slightly when the key size increases above 25, and even then remains within 0.05% of the same number of keys as the cryptographic hashers, whereas the naive First-n approach is 32% behind any of the others.

We find that as more hash keys are used, fewer collisions require less work to invert the hash table, resulting in faster extraction times. Figure B.2. We also measured the computational time required to hash the input sequences under each hash function, as shown in Figure B.3.

The extraction times when using MD5 versus adaptive hash functions start to diverge slightly as pattern size increases. This divergence is a direct consequence of the divergence in hash key space utilization between the adaptive hasher and MD5 for very large patterns, as shown in Figure B.4. The extent to which this divergence occurs and where it starts is in fact tunable via a parameter in the hash function generator genetic algorithm; however, for such long patterns we consider this divergence reasonable, as it has a minimal impact on the extraction time but a great savings on the hash time.

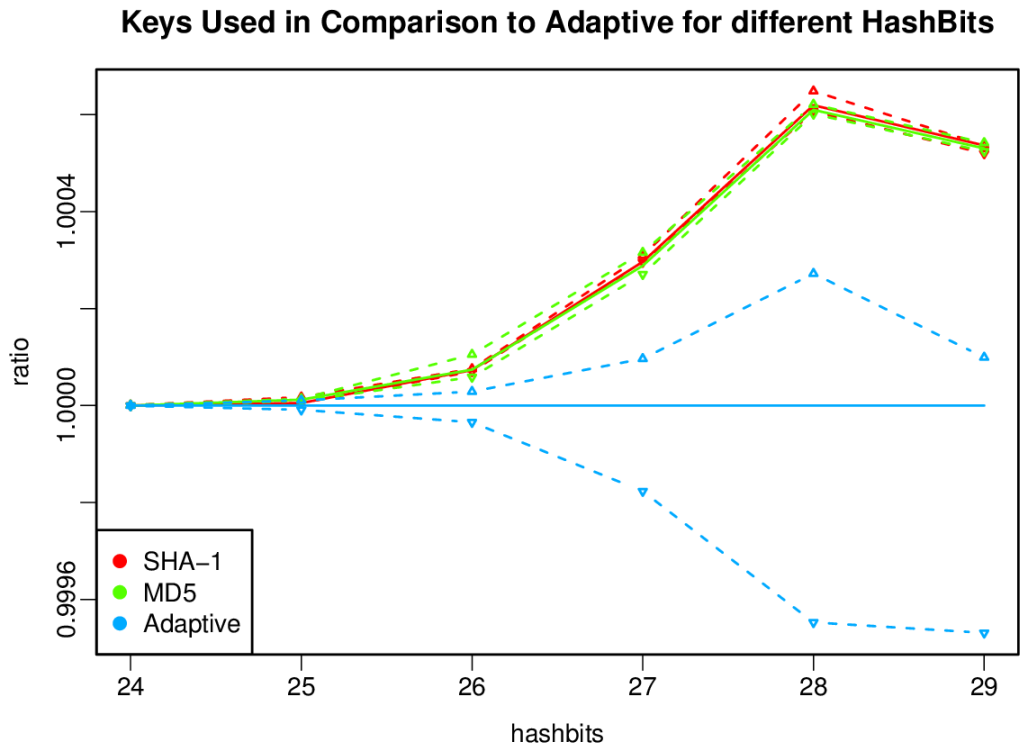

Figure B.1: This graph doesn't include First-N in order to examine, and adaptive hash results to examine the minute difference between Adaptive, SHA-1, and MD5. Only for large hash keys (high values of hashbits) does adaptive diverge significantly from SHA-1 and MD5, and even then the difference is minuscule.

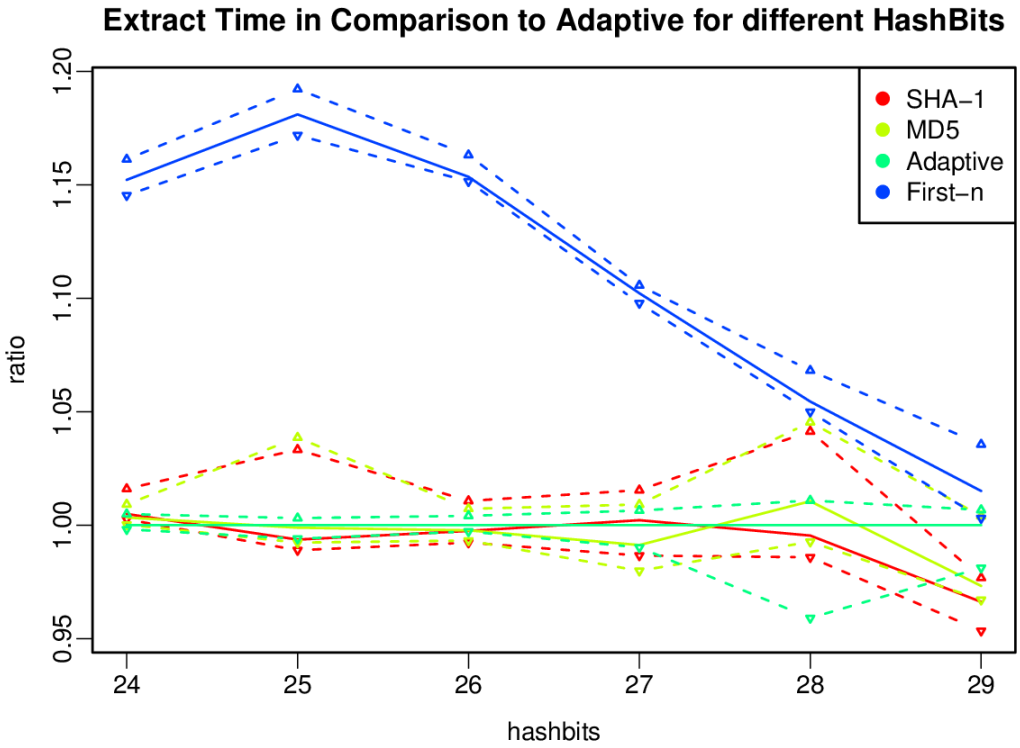

Figure B.2: This graph shows the relative time required to extract matching seed sets from the hash table under different hash functions compared to the median time required our adaptive hash function. The solid line shows the median of all trials while the dashed lines show the first and third quartiles.

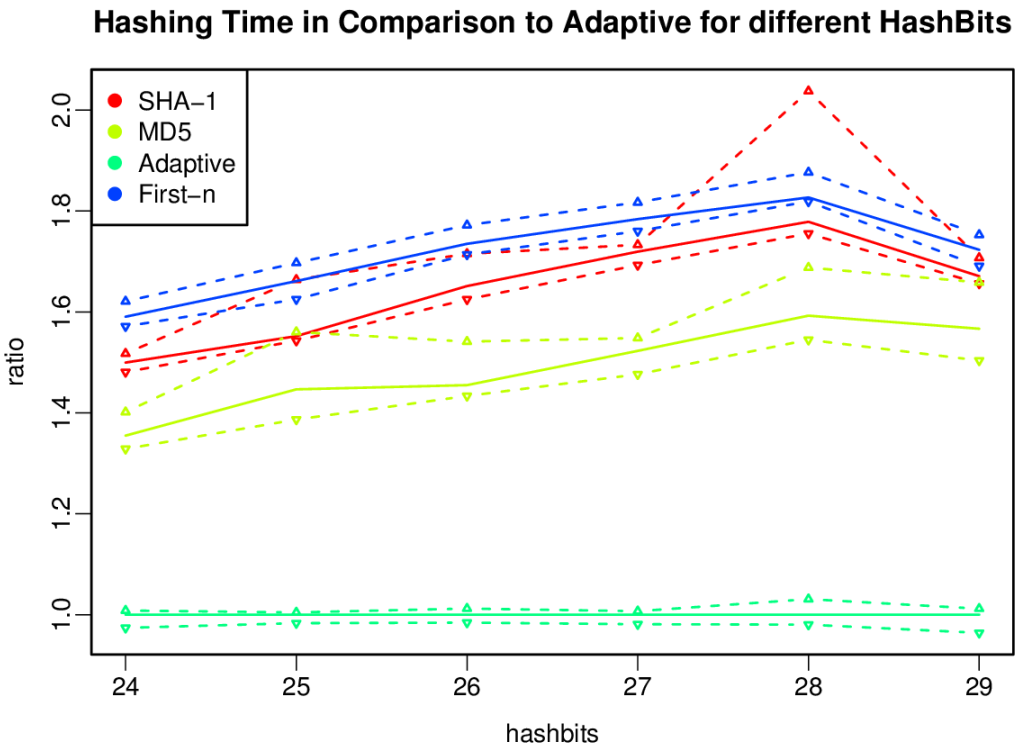

Figure B.3: Here we examine the difference in time required to compute hashes store each (K, V) pair at different hashbits settings, again compared to our adaptive hash method. It's interesting to note that the naive First-n approach performs more poorly than even the slowest cryptographic hasher.

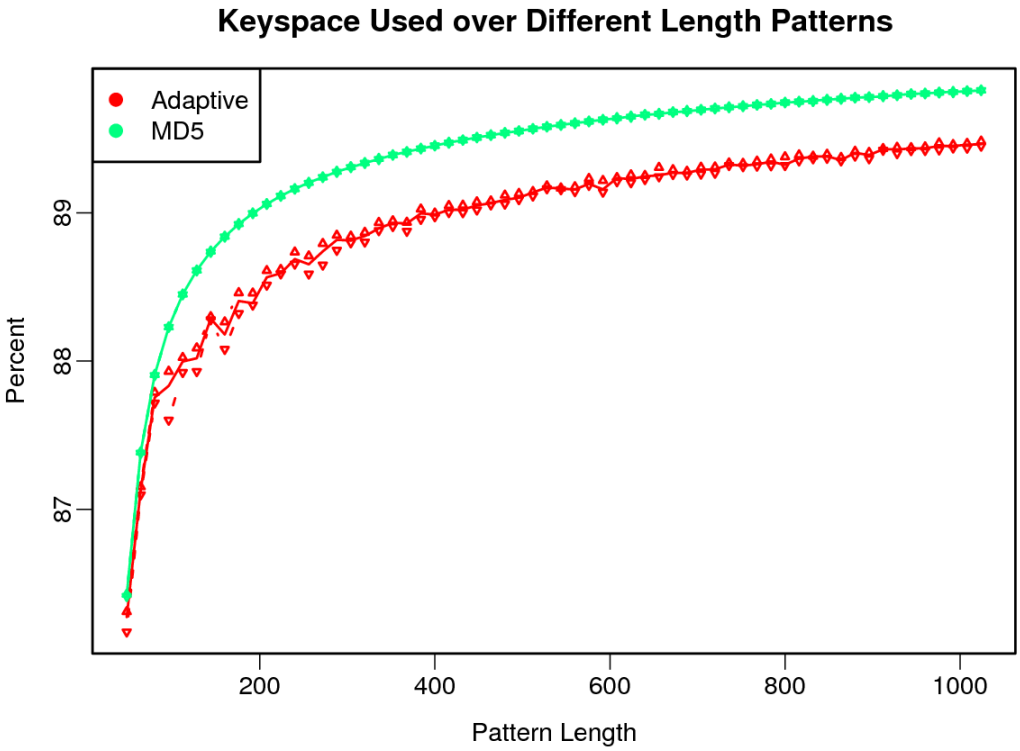

Figure B.4: This graph shows the percent of possible hash keys produced by Adaptive and MD5 hash functions when hashing Human and Mouse X chromosomes. The number of hash keys possible increases with pattern length because the number of observed unique seeds increases. Our adaptive hash algorithm keeps up with MD5 even for extremely long patterns.

B.1.2 Implementation Details

Due to the wide variety of applications to which Murasaki is applied, we have built in a wide variety of user-tunable parameters to optimize Murasaki for various special purposes (such as hashing only every k bases as in BLAT (Kent, 2002) and PASH (Ken J. Kalafus, 2004), defining alternative scoring/filtering parameters, or changing resource allocation when run in parallel). Most of the user-tunable parameters are determined automatically based on the input data and/or the user's computer environment. The main user-tunable parameter determining run time is the size of hash keys to use (which directly determines the hash table size), which we will call the "hashbits" parameter.

When storing hash keys, because K is the product of the location L and the "spaced seed pattern" parameter, K is determined implicitly by L and thus is never actually stored, but simply calculated as needed based on L.

B.1.3 Data structures

There are two important data structures within Murasaki.

1. The hash table structure
2. The anchor data structure

First, the hash table itself can be structured to use either chaining or open addressing. Under chaining, each entry of the hash table can contain values corresponding to multiple keys. When a collision occurs, where $K_i \neq K_j \wedge H(K_i) = H(K_j)$, then both corresponding L_i and L_j are stored at the same entry within the hash table, and disambiguated by some other means specific to the table entry structure. Under open addressing, the hash generated by $H(K_i)$ indicates the first place to *probe* for K_i. If some (K_j, L_j) pair is already stored there such that $K_i \neq K_j$ (i.e., a collision has occurred), then subsequent addresses are *probed* until either a matching K is found, or an empty address is found. There are many variations on how to probe, and Murasaki implements two (linear and quadratic probing (Knuth, 1973)), but the idea remains the same. Open Addressing offers the advantage that there is no *sort* required to disambiguate non-matching seeds from within a single hash table entry. However a limitation of this design is that it is impossible to store more seeds than there are entries in the hash table. Murasaki automatically chooses whether to use chaining or open addressing based on the size of the input sequences (and thus the number of potential seeds) and the hash table size available.

The second key data structure is that for storing anchors themselves. Anchors are stored as a set of intervals indicating the beginning and end of the

region anchored on each input sequence. These intervals are stored in Interval Trees (Cormen, Leiserson, and Rivest, 1990). Interval Trees provide a simple $O(\log A)$ method (where A is the number of anchors) to find overlapping regions, thereby facilitating the merging of anchors.

B.1.4 Hash function fitness

Determining the exact entropy content of a given hash function is a difficult problem with limited rewards compared with a good approximation; therefore, Murasaki employs an ad hoc method to estimate the entropy of the hash function. A bipartite graph is created consisting of entropy source nodes (the unmasked bases in the spaced seed pattern) and entropy sink nodes (each pair of bits in the hash key) where an edge is drawn if that base in the seed affects that bit in the hash. Each source node is allocated one unit of entropy (approximating the optimal 2 bits per base measure) that is then divided equally to weight the exiting edges. The entropy at each sink is then estimated as the summed weight of incoming edges, optionally modified by a "correlation penalty" in the range $(0, .2]$. The correlation penalty decreases inversely with the mean distance between source node pairs, with a maximum value of 0.2 assigned for adjacent bases. A correlation penalty like this is appropriate where input sequences are expected to conform to a Markov process (as is the case for DNA), but this may not be the case for all sequences. Finally, the estimated entropy is balanced against the computational cost of the function to determine the final fitness score.

B.2 Results

B.2.1 Pattern selection

We used a relatively arbitrary method for selecting the 24 base pattern (101111110101110111110011) used for tests comparing Murasaki to BLASTZ. We compared human and mouse X chromosomes using random patterns with lengths from 16 to 128. Patterns with lengths near 24 had sensitivity and specificity characteristics near that of BLASTZ while maintaining fast computational speed. We arbitrarily chose the highest scoring of our randomly generated length 24 patterns. We recognize that pattern choice is a complicated and important factor in the performance of ours and any other pattern-based homology search algorithm, and don't claim to have offered any better solutions to the problem of pattern choice, however our algorithm does provide a means run accurate searches using patterns selected by any

method. We hope this provides a useful tool for experimenting with different pattern selection methods, and will look at ways to refine pattern selection in future work.

B.2.2 Murasaki Runtime Parameters

We also used the options "–mergefilter=100 and –scorefilter=6000" which is roughly equivalent to BLASTZ's "M=100 K=6000" options. "Mergefilter" limits the number of anchors that can be derived from any one seed to the number specified; any seeds which exceed this limit are tagged "repeats" and their locations are output separately. "Scorefilter" requires that all anchors have a minimum ungapped pairwise alignment score of at least the given threshold. For the exact meaning and usage of "mergefilter" and "scorefilter" see Murasaki's documentation. Furthermore, because the default TBA behavior is to pass its BLASTZ output through a program that removes all but the highest scoring regions among overlapping regions in a pair-wise comparison (similar to AxtBest in (Schwartz et al., 2003)), to provide a fair comparison, we filtered Murasaki's anchors the same way using the `align-best` tool provided in the Murasaki distribution.

The "mergefilter" filter alone can prevent the combinatorially explosive consequences of repeats, however in our mammalian sequence tests we use repeat masked sequences to reduce the amount of sequence hashed and stored in memory. We used the RepeatMasker (Smit, R, and Green, 1996-2004) masked sequences in release 53 of the Ensembl genome database (Hubbard et al., 2002). The genome sizes and fraction masked by RepeatMasker is shown in Table B.2.2.

B.2.3 Pairwise Multiz with Roast

We also tested another alternative from the TBA package called Roast which appears to implement the method described in (Miller et al., 2007) which builds a multiple alignment based on pairwise comparisons between a reference sequence and all other sequences, thus in theory requiring time linear with respect to the number of sequences, similar to Murasaki. However due to an apparent bug in the implementation, Roast actually is actually worse than TBA in some cases. The bug causes comparisons of each sequence to itself to be included in the set of pairwise comparisons performed for each alignment (for example "human-human" in a comparison of "human and mouse"). This is entirely unnecessary for the Multiz portion of the computation, but generally takes more than any other pair because of the large number of matches.

This bug is fairly simple to fix, however may be beyond the reach of biologists without some programming ability, therefore we tested both versions. The results are shown in Figure B.5. Because BLASTZ cannot handle whole genomes in a single pass, input has to be broken up into chromosome sized fragments. This means that what was a comparison of "human x mouse" in Murasaki, becomes a comparison of "human-1 x mouse-1 + human-1 x mouse-2 + ... human-1 x mouse-M + human-2 x mouse-1 + human2 x mouse-2 + ... human-M x mouse-M." Each comparison is shorter (N/M in size), however we now do M^2 times as many, requiring $O(NM)$ time. However, because M is forced by the constraints of the system to be N/S where S is the maximum size the computer system can handle, $O(M) = O(N/S) = O(N)$ and therefore the asymtotic time requirements of Roast must be $O(N^2)$. The behavior of Roast on small sequences, on the other hand, is nearly identical to Murasaki, and is shown in Figure B.6.

Table B.1: Genome Sizes

Species	Total Bases (MB)	Masked Bases (MB)	Fraction masked
Human	2855.344	1434.406	0.502
Chimp	2752.356	1402.532	0.510
Rhesus	2646.263	1375.097	0.520
Orangutan	2722.968	1348.973	0.495
Mouse	2558.509	1443.026	0.564
Rat	2477.054	1386.647	0.560
Dog	2309.875	1367.003	0.592
Cow	2466.956	1319.883	0.535

This table shows the sequence sizes used in the mammalian whole genome and comparisons and the respective fractions masked by RepeatMasker (Smit, R, and Green, 1996-2004).

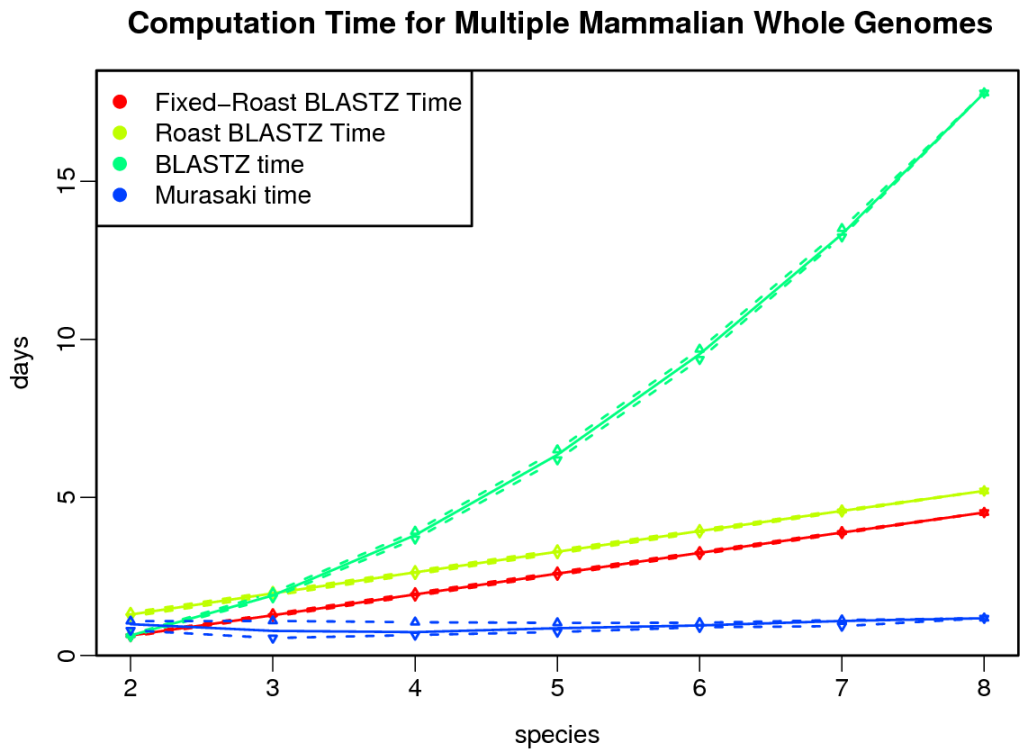

Figure B.5: This graph shows the median CPU time in days required to anchor different numbers of mammalian whole genomes using TBA, Murasaki, and the patched and unpatched versions of Roast. The times for TBA and Roast include only the time spent on pairwise BLASTZ comparisons. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles.

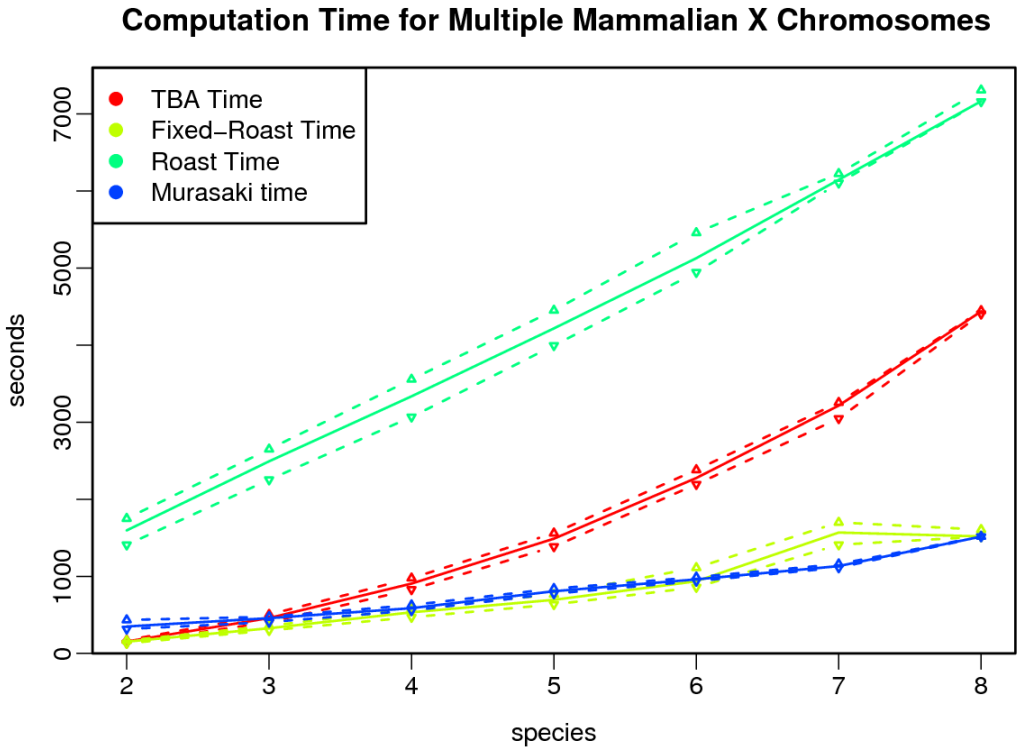

Figure B.6: This graph compares the computational time required to compare multiple X mammalian X chromosomes using Murasaki and the BLASTZ components of TBA, Roast, and our patched version of Roast. Because TBA requires all pairwise comparisons of the genomes under alignment, the time required for TBA grows quadratically, while Murasaki's time is near linear. The solid line represents the median of all tests for that number of species, while the dashed lines represent the first and third quartiles.

Appendix C

Appendix: Samscope

C.1 BIP File format

C.1.1 Description

Samscope uses a binary format for storing MIP maps of data like coverage. This makes it possible view data while zoomed out to any scale without sampling each individual component base. We call our format "BIP" for "*B*inary m*IP* map". This data is layed out in a flat format with indices for accessing data as fast as possible. At present, BIP we implement no compression, because we assume that NGS users will have a lot of spare disk space. File system compression (as in wikipedia:ZFS, wikipedia:btrfs, and wikipedia:ReiserFS) works very well if you really want compression.

The BIP file format supports multiple layers per BIP file, however the current version Samscope outputs each layer to a separate file for easier management by users.

C.1.2 Specification

The file format consists of a fixed sized header, followed by a variable size header (describing things like names of the content), followed by a the layer data itself. Strings are expressed as "length" (expressed as a SizeUnit) followed by the string itself (exactly "length" bytes).

Fixed header

Field	Type	Size or count	Description
magic	bytes	8	Contains "\x04\-0am\0BIP" (Nothing good happens at 4 am). Identifies file type.
version	uint8	1	Format version. Parsers are not required to support all versions, but should check this value and error gracefully if they can't support this version. This specification refers to version 1 and 2.
byte order	uint16	2	Contains 0x1234. Denotes byte order of creating machine. Parsers are not *required* to read files using byte orders or data sizes unsupported by their architecture (but they should check using this header data and error gracefully if they can't read the BIP).
SizeUnit	uint8	1	How many bytes to use in expressing data lengths and indices.

(continued below)

(Fixed header continued)

Field	Type	Size or count	Description
CoordSize[2]	uint8	1	How many bytes to use in expressing value coordinates (should be large enough to express total target length).
CounterSize[2]	uint8	1	How many bytes to use in expressing data values.
RangeUnitSize[2]	uint8	1	How many bytes to use in expressing counter ranges (should be greater than CounterSize).
headerLength	uint	SizeUnit	Length of full header (including both fixed and variable parts).
globalHistoDetail	uint	Counter∗	Maximum number of values per MIP column.

Note: the CoordSize, CounterSize, and RangeUnitSize fields marked with [2] are part of BIP format 2, and simply omitted in version 1. There are no other differences between versions 1 and 2.

Variable length header

Field	Type	Size	Description
targetCount	uint	SizeUnit	How many targets in this BIP?
∗	targetDescription	variable	name and length of N targets (repeats *targetCount* times)
layerCount	uint	SizeUnit	How many layers in this BIP?
∗	layerDescription	variable	name and range of N layers (repeats *layerCount* times)
endFlag	char	9	"!!!@@@!!!" nice ASCII readable marker to signal end of variable header (just a sanity check, really).

Target Description

Field	Type	Size	Description
startPoint	uint	Coord	start point of target
name	string∗	variable	name of target

Layer Description

Field	Type	Size	Description
name	string∗	variable	name of the layer
$range_i$	pair < RangeUnit, RangeUnit >	RangeUnit × 2	highest and lowest value within target *i*.
$range_{i+1}$	...	...	repeats for each target

Layer data

Each layer includes a "base coverage" data segment and $\lceil \log_2 N \rceil$ MipMap segments (for N bases of target sequence).

The first "base coverage" data segment has 1 value for each base. This is listed completely flat, 1 Counter per base. The subsequent MipMaps are variable in length, depending on the number of distinct values per column, with an index describing the start of each column.

Each segment starts immediately after the end of the previous one, thus their start addresses are calculated by first the end of the variable header data, then by the end of the first base coverage, then by the end of the MipMap data, and so on.

Base coverage

The base coverage starts at the end of the variable header and continues for exactly the number of total bases as determined by the sum of target lengths.

Field	Type	Size	Description
baseCoverage	Counter[ ]	Counter × totalLength	array of data values, 1 per base.

MIP Maps

Each MIP map coverages half the columns as the previous iteration (rounding up). MIP maps continue until only 1 column remains (this MIP map too is stored, however it is the last). Each mip map is preceded by its index (without which the total length would be unknown). The index indicates the number of HistoElem pairs (literally just 2 Counters) in to the MIP at which each column starts. In other words:

```
//assume we have a MIP map of HistoElem's start at map
HistoElem *map="first byte after end of last segment";
//Column i can be defined as going from:
HistoElem *begin=map+index[i];
//and ending at:
HistoElem *end=map+index[i+1];
```

Note that the last value in the index merely points to the *end* of the array (and next BIP segment, if one exists). Therefore there is 1 more index entry than there are columns. The "HistoElem" type merely consists of a pair of Counters, the first referring to the number of times the value is repeated in the column, and the second refers to the value being repeated.

Field	Type	Size	Description
index	SizeUnit[ ]	number of remaining columns +1	Index of HistoElems in the MIP map
MIPmap	HistoElem[ ]	variable	Array of Histogram elements.

Bibliography

Alekseyev, Max A. and Pavel A. Pevzner (Nov. 2007). "Are There Rearrangement Hotspots in the Human Genome?" In: *PLoS Computation Biology* 3.11, e209. DOI: 10.1371/journal.pcbi.0030209.

Altschul, S. F. et al. (1990). "Basic local alignment search tool". In: *Journal of Molecular Biology* 215, pp. 403–410.

Bejerano, Gill et al. (2004). "Ultraconserved Elements in the Human Genome". In: *Science* 304.5675, pp. 1321–1325. DOI: 10.1126/science.1098119. eprint: http://www.sciencemag.org/cgi/reprint/304/5675/1321.pdf. URL: http://www.sciencemag.org/cgi/content/abstract/304/5675/1321.

Bentley, D. R. et al. (2008). "Accurate whole human genome sequencing using reversible terminator chemistry". In: *Nature* 456, pp. 53–59.

Blanchette, Mathieu et al. (2004). "Aligning Multiple Genomic Sequences With the Threaded Blockset Aligner". In: *Genome Research* 14.4, pp. 708–715. DOI: 10.1101/gr.1933104. eprint: http://genome.cshlp.org/content/14/4/708.full.pdf+html. URL: http://genome.cshlp.org/content/14/4/708.abstract.

Bourque, Guillaume and Pavel A. Pevzner (2002). "Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species". In: *Genome Research* 12.1, pp. 26–36. eprint: http://www.genome.org/cgi/reprint/12/1/26.pdf. URL: http://www.genome.org/cgi/content/abstract/12/1/26.

Brudno, Michael et al. (2003). "LAGAN and Multi-LAGAN: Efficient Tools for Large-Scale Multiple Alignment of Genomic DNA". In: *Genome Research* 13.4, pp. 721–731. DOI: 10.1101/gr.926603. eprint: http://www.genome.org/cgi/reprint/13/4/721.pdf. URL: http://www.genome.org/cgi/content/abstract/13/4/721.

Burrows, M. and D.J. Wheeler (1994). *A block-sorting lossless data compression algorithm*. Tech. rep. 124. Palo Alto, CA: Digital Equipment Corporation, Systems Research Center.

Cormen, T., C. Leiserson, and R. Rivest (1990). *Introduction to Algorithms*. MIT Press.

Csuros, M. and B. Ma (2007). "Rapid homology search with neighbor seeds". In: *Algorithmica* 48.2, pp. 187–202.

Darling, Aaron C.E. et al. (2004). "Mauve: Multiple Alignment of Conserved Genomic Sequence With Rearrangements". In: *Genome Research* 14.7, pp. 1394–1403. DOI: 10.1101/gr.2289704. eprint: http://www.genome.org/cgi/reprint/14/7/1394.pdf. URL: http://www.genome.org/cgi/content/abstract/14/7/1394.

David, M. et al. (2011). "SHRiMP2: sensitive yet practical SHort Read Mapping". In: *Bioinformatics* 27, pp. 1011–1012.

Delcher, AL et al. (1999). "Alignment of whole genomes". In: *Nucleic Acids Research* 27.11, pp. 2369–2376. DOI: 10.1093/nar/27.11.2369. eprint: http://nar.oxfordjournals.org/cgi/reprint/27/11/2369.pdf. URL: http://nar.oxfordjournals.org/cgi/content/abstract/27/11/2369.

Dewey, C. N. et al. (2006). "Parametric alignment of Drosophila genomes". In: *PLoS Computational Biology* 2, e73.

Enard, W. et al. (2002). "Molecular evolution of FOXP2, a gene involved in speech and language". In: *Nature* 418.6900, pp. 869–872.

Farach, Martin et al. (1995). "On the entropy of DNA: algorithms and measurements based on memory and rapid convergence". In: *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. San Francisco, California, United States: Society for Industrial and Applied Mathematics, pp. 48–57. ISBN: 0-89871-349-8.

Farnham, P.J. (2009). "Insights from genomic profiling of transcription factors". In: *Nature Reviews Genetics* 10.9, pp. 605–616.

Ferragina, P. and G. Manzini (2000). "Opportunistic data structures with applications". In: *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, pp. 390–398.

Fisher, S.E. et al. (1998). "Localisation of a gene implicated in a severe speech and language disorder". In: *Nature genetics* 18.2, pp. 168–170.

Gibbs, R. A. et al. (2004). "Genome sequence of the Brown Norway rat yields insights into mammalian evolution." In: *Nature* 428.6982, pp. 493–521.

Glenn, T.C. (2011). "Field guide to next-generation DNA sequencers". In: *Molecular Ecology Resources* 11.5, pp. 759–769.

Hachiya, Tsuyoshi et al. (2009). "Accurate identification of orthologous segments among multiple genomes". In: *Bioinformatics* 25.7, pp. 853–860. DOI: 10.1093/bioinformatics/btp070. eprint: http://bioinformatics.oxfordjournals.org/cgi/reprint/25/7/853.pdf. URL: http://

bioinformatics.oxfordjournals.org/cgi/content/abstract/25/7/853.

Han, T. and S. Parameswaran (2002). "SWASAD: an ASIC design for high speed DNA sequence matching". In: *Design Automation Conference, 2002. Proceedings of ASP-DAC 2002. 7th Asia and South Pacific and the 15th International Conference on VLSI Design. Proceedings.* IEEE, pp. 541–546.

Hubbard, T. et al. (2002). "The Ensembl genome database project". In: *Nucleic Acids Research* 30.1, pp. 38–41.

Karolchik, D. et al. (2003). "The UCSC Genome Browser Database". In: *Nucleic Acids Research* 31.1, pp. 51–54. DOI: 10.1093/nar/gkg129. eprint: http://nar.oxfordjournals.org/cgi/reprint/31/1/51.pdf. URL: http://nar.oxfordjournals.org/cgi/content/abstract/31/1/51.

Kemena, Carsten and Cedric Notredame (2009). "Upcoming challenges for multiple sequence alignment methods in the high-throughput era". In: *Bioinformatics* 25.19, pp. 2455–2465. DOI: 10.1093/bioinformatics/btp452. eprint: http://bioinformatics.oxfordjournals.org/cgi/reprint/25/19/2455.pdf. URL: http://bioinformatics.oxfordjournals.org/cgi/content/abstract/25/19/2455.

Ken J. Kalafus Andrew R. Jackson, Aleksandar Milosavljevic (2004). "Pash: Efficient Genome-Scale Sequence Anchoring by Positional Hashing". In: *Genome Research* 14, pp. 672–678. DOI: 10.1101/gr.1963804.

Kent, W. J. et al. (2010). "BigWig and BigBed: enabling browsing of large distributed datasets". In: *Bioinformatics* 26, pp. 2204–2207.

Kent, W. James (2002). "BLAT—The BLAST-Like Alignment Tool". In: *Genome Research* 12.4, pp. 656–664. DOI: 10.1101/gr.229202. eprint: http://www.genome.org/cgi/reprint/12/4/656.pdf. URL: http://www.genome.org/cgi/content/abstract/12/4/656.

Kiełbasa, S.M. et al. (2011). "Adaptive seeds tame genomic sequence comparison". In: *Genome Research* 21.3, pp. 487–493.

Kish, Laszlo B. (2002). "End of Moore's law: thermal (noise) death of integration in micro and nano electronics". In: *Physics Letters A* 305.3-4, pp. 144–149. ISSN: 0375-9601. DOI: DOI:10.1016/S0375-9601(02)01365-8. URL: http://www.sciencedirect.com/science/article/B6TVM-475B5CK-1/2/3e881305a670968756f015af1fc1f22f.

Knuth, Donald E. (1973). *Sorting and Searching, volume 3 of The art of computer programming.* Addison Wesley.

Koonin, E.V. (2005). "Orthologs, paralogs, and evolutionary genomics". In: *Annual Review of Genetics* 39, pp. 309–338.

Kulla, F. and P. Sanders (2007). "Scalable parallel suffix array construction". In: *Parallel Computing* 33.9, pp. 605–612.

Langmead, B. et al. (2009). "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome". In: *Genome Biology* 10, R25.

Li, H. and R. Durbin (2009). "Fast and accurate short read alignment with Burrows-Wheeler transform". In: *Bioinformatics* 25, pp. 1754–1760.

Li, H., J. Ruan, and R. Durbin (2008). "Mapping short DNA sequencing reads and calling variants using mapping quality scores". In: *Genome Research* 18, pp. 1851–1858.

Li, H. et al. (2009). "The Sequence Alignment/Map format and SAMtools". In: *Bioinformatics* 25, pp. 2078–2079.

Li, I.T.S., W. Shum, and K. Truong (2007). "160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (FPGA)". In: *BMC Bioinformatics* 8.1, p. 185.

Liolios, K. et al. (2008). "The Genomes On Line Database (GOLD) in 2007: status of genomic and metagenomic projects and their associated metadata". In: *Nucleic Acids Research* 36, pp. D475–479.

Ma, Bin, John Tromp, and Ming Li (2002). "PatternHunter: faster and more sensitive homology search". In: *Bioinformatics* 18.3, pp. 440–445.

Maier, D. (1978). "The complexity of some problems on subsequences and supersequences". In: *Journal of the ACM (JACM)* 25.2, pp. 322–336.

Mathee, Kalai et al. (2008). "Dynamics of Pseudomonas aeruginosa genome evolution". In: *Proceedings of the National Academy of Sciences* 105.8, pp. 3100–3105. DOI: 10.1073/pnas.0711982105. eprint: http://www.pnas.org/content/105/8/3100.full.pdf+html. URL: http://www.pnas.org/content/105/8/3100.abstract.

Miller, W. et al. (2007). "28-way vertebrate alignment and conservation track in the UCSC Genome Browser". In: *Genome Research* 17, pp. 1797–1808.

Milne, I. et al. (2010). "Tablet–next generation sequence assembly visualization". In: *Bioinformatics* 26, pp. 401–402.

Moore, G.E. et al. (1965). "Cramming more components onto integrated circuits". In: *Proceedings of the IEEE* 38.8, p. 114.

Mortazavi, A. et al. (2008). "Mapping and quantifying mammalian transcriptomes by RNA-Seq". In: *Nature Methods* 5.7, pp. 621–628.

National Institute of Standards and Technology (NIST) (2002). *FIPS-180-2: Secure Hash Standard*. URL: http://www.itl.nist.gov/fipspubs/.

Nicol, J. W. et al. (2009). "The Integrated Genome Browser: free software for distribution and exploration of genome-scale datasets". In: *Bioinformatics* 25, pp. 2730–2731.

Ohlebusch, E. and S. Kurtz (2008). "Space efficient computation of rare maximal exact matches between multiple sequences". In: *Journal of Computational Biology* 15, pp. 357–377.

Osana, Yasunori, Kris Popendorf, and Yasubumi Sakakibara (In preparation). *GMV: Interactive Rendering of Multiple Alignments*. URL: `http://murasaki.dna.bio.keio.ac.jp`.

Pagani, I. et al. (2012). "The Genomes OnLine Database (GOLD) v. 4: status of genomic and metagenomic projects and their associated metadata". In: *Nucleic Acids Research* 40.D1, pp. D571–D579.

Park, P.J. (2009). "ChIP–seq: advantages and challenges of a maturing technology". In: *Nature Reviews Genetics* 10.10, pp. 669–680.

Pearson and Lipman (1988). "Improved Tools for Biological Sequence Comparison". In: *Proceedings of the National Academy of Sciences*. Vol. 85, pp. 24444–24448.

Pepke, S., B. Wold, and A. Mortazavi (2009). "Computation for ChIP-seq and RNA-seq studies". In: *Nature Methods* 6, pp. 22–32.

Pevzner, P.A. and H. Tang (2001). "Fragment assembly with double-barreled data". In: *Bioinformatics* 17.suppl 1, S225–S233.

Pevzner, Pavel and Glenn Tesler (2003). "Genome Rearrangements in Mammalian Evolution: Lessons From Human and Mouse Genomes". In: *Genome Research* 13.1, pp. 37–45. DOI: `10.1101/gr.757503`. eprint: `http://www.genome.org/cgi/reprint/13/1/37.pdf`. URL: `http://www.genome.org/cgi/content/abstract/13/1/37`.

Pollard, K.S. et al. (2006). "An RNA gene expressed during cortical development evolved rapidly in humans". In: *Nature* 443.7108, pp. 167–172.

Preparata, F. P., L. Zhang, and K. W. Choi (Nov. 2005). "Quick, practical selection of effective seeds for homology search." In: *Journal of Computational Biology* 12.9, pp. 1137–1152.

Quinlan, S. and S. Dorward (2002). "Venti: a new approach to archival storage". In: *Proceedings of the FAST 2002 Conference on File and Storage Technologies*. Vol. 4.

Rivest, R.L. (1992). *The MD5 message-digest algorithm*. URL: `http://tools.ietf.org/html/rfc1321`.

Robinson, J. T. et al. (2011). "Integrative genomics viewer". In: *Nature Biotechnology* 29, pp. 24–26.

Rogers, Y.H. and J.C. Venter (2005). "Genomics: massively parallel sequencing". In: *Nature* 437.7057, pp. 326–327.

Rumble, Stephen M. et al. (May 2009). "SHRiMP: Accurate Mapping of Short Color-space Reads". In: *PLoS Computational Biology* 5.5, e1000386. DOI: `10.1371/journal.pcbi.1000386`. URL: `http://dx.doi.org/10.1371%2Fjournal.pcbi.1000386`.

Russ, John C. (2007). *The Image Processing Handbook*. Fifth. Boca Raton : CRC/Taylor and Francis. ISBN: 0849372542.

Sanger, F., A.R. Coulson, et al. (1975). "A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase". In: *Journal of Molecular Biology* 94.3, pp. 441–448.

Schatz, M.C., J. Witkowski, and W.R. McCombie (2012). "Current challenges in de novo plant genome sequencing and assembly". In: *Genome Biology* 13.4, pp. 1–7.

Schwartz, Scott et al. (2003). "Human-Mouse Alignments with BLASTZ". In: *Genome Research* 13.1, pp. 103–107. DOI: 10.1101/gr.809403. eprint: http://www.genome.org/cgi/reprint/13/1/103.pdf. URL: http://www.genome.org/cgi/content/abstract/13/1/103.

Shang, W. H. et al. (2010). "Chickens possess centromeres with both extended tandem repeats and short non-tandem-repetitive sequences". In: *Genome Research* 20, pp. 1219–1228.

Simpson, Jared T. et al. (2009). "ABySS: A parallel assembler for short read sequence data". In: *Genome Research* 19.6, pp. 1117–1123. DOI: 10.1101/gr.089532.108. eprint: http://genome.cshlp.org/content/19/6/1117.full.pdf+html. URL: http://genome.cshlp.org/content/19/6/1117.abstract.

Smit, AFA, Hubley R, and P. Green (1996-2004). *RepeatMasker Open-3.0*. URL: http://www.repeatmasker.org.

Smith, Temple F. and Michael S. Waterman (1981). "Identification of Common Molecular Subsequences". In: *Journal of Molecular Biology* 147, pp. 195–197.

Tabus, I. and G. Korodi (2008). "Genome compression using normalized maximum likelihood models for constrained Markov sources". In: *Information Theory Workshop, 2008. ITW '08. IEEE*, pp. 261–265. DOI: 10.1109/ITW.2008.4578663.

Teer, J.K. et al. (2010). "Systematic comparison of three genomic enrichment methods for massively parallel DNA sequencing". In: *Genome Research* 20.10, pp. 1420–1431.

Thompson, Julie D., Desmond G. Higgins, and Toby J. Gibson (1994). "CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice". In: *Nucleic Acids Research* 22.22, pp. 4673–4680. DOI: 10.1093/nar/22.22.4673. eprint: http://nar.oxfordjournals.org/cgi/reprint/22/22/4673.pdf. URL: http://nar.oxfordjournals.org/cgi/content/abstract/22/22/4673.

Tuzun, E. et al. (2005). "Fine-scale structural variation of the human genome". In: *Nature Genetics* 37.7, pp. 727–732.

Vallender, E. J. et al. (2006). "SPEED: a molecular-evolution-based database of mammalian orthologous groups". In: *Bioinformatics* 22, pp. 2835–2837.

Waterston, R. H. et al. (2002). "Initial sequencing and comparative analysis of the mouse genome". In: *Nature* 420, pp. 520–562.

Williams, L. (Feb. 1983). "Pyramidal Parametrics". In: *Computer Graphics*, pp. 1–11.

Zerbino, D.R. and E. Birney (2008). "Velvet: algorithms for de novo short read assembly using de Bruijn graphs". In: *Genome Research* 18.5, pp. 821–829.

Zhu, Z. et al. (2012). "MGAviewer: A desktop visualization tool for analysis of metagenomics alignment data". In: *Bioinformatics*.

List of Figures