Efficient Genomic Interval Intersection Algorithms

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Ryan M Layer

May 2014

Abstract

The comparison of diverse genomic datasets is fundamental to understanding genome biology. Because individual experiments typically focus on a single genomic feature (e.g., genes, regulatory elements, genetic variants), the discovery and characterization of higher-level functions requires the integration of many different experimental results. For example, certain genomic sequences called enhancers have been shown to modulate gene expression. Since these sequences also tend to have specific chemical markers (histone marks) and can be targets of sequence-specific DNA binding proteins (transcription factors), comparing the set of sequences with certain histone marks to the set that are bound by transcription factors can lead to the discovery of novel enhancers. This is a difficult problem, especially considering that it is common for these types of experiments to identify millions of sequences. However, it can be simplified by the use of a reference genome, which is a representative nucleotide sequence for the chromosomes in an organism's genome. A reference provides the coordinate system for mapping a given sequence to an interval based on the relative position of its matching subsequence. By mapping the sequences in feature sets to intervals, the comparison between sets is reduced to an interval set intersection problem. That is, if two features intersect, then they share common genomic sequences and can thus possibly have a biologically relevant relationship.

Given the utility of efficient interval intersection in genomic analysis, especially in the face of every-larger data sets, this thesis introduces three algorithms that improve upon the state-of-the-art. The Binary Interval Search (BITS) algorithm is an optimal algorithm for counting the number of intersections between two sets that is well suited for parallel processing on a GPU. By counting intersections without full enumeration (a fundamentally harder problem), BITS enables the large-scale comparison of thousands of genomic features on a single machine that would have otherwise required a cluster of over 100 nodes. The split-then-sweep algorithm extends the basic concepts introduced by BITS to the consideration of many genomic interval sets. Sets are partitioned into independent slices, which in many cases reduces the number of the intervals that must be considered, and in all cases exposes a significant amount of parallelism. Slicing improved sequential runtimes and parallel runtimes by a factor of 2 and 19, respectively. The LUMPY algorithm uses interval intersection as a platform to identify differences in genome structure from high-throughput sequencing data. Unlike any other algorithm in this domain, LUMPY is able to jointly consider many different types of evidence. This greatly increases its sensitivity, especially in cases where only a fraction of cells contain a specific variation, which is typical in cancer genomes. These algorithms significantly advance the field by greatly expanding the data that can be considered in the analysis of genome intervals and high-throughput sequencing data.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Ryan M Layer

This dissertation has been read and approved by the Examining Committee:

Gabriel Robins, Adviser

James Cohoon, Committee Chair

Kevin Skadron

Ira Hall

Aaron Quinlan

Accepted for the School of Engineering and Applied Science:

James Aylor, Dean, School of Engineering and Applied Science

May 2014

To my wife Lindsey, for making me better.

Acknowledgments

I would like to thank all of my advisors: Professors Gabriel Robins, Ira Hall, and Aaron Quinlan. Gabriel for guiding me into the field of computational biology, helping me through all the twists, turns, and bumps of my journey, and his constant focus on what is important. Ira for helping me change directions at exactly the right time, always knowing why my ideas wouldn't work then helping me to fix them, and helping me to realize where and how I could make a difference. Aaron for showing interest in the little script that eventually became my main research topic, helping me to appreciate that usability and documentation are important, and including me in his crazy ideas. They have always treated me like a colleague, and I expect to continue to rely on them throughout my career.

Thank you to Professor Anindya Dutta for his guidance and patience during my transition into genomics and his pep talks when I was sure this was a bad idea. I am also grateful to all of the members of Dutta lab, especially Dr. Neerja Karnani, Dr. Migon Keaton, and Dr. Pankaj Kumar, who spent many hours helping me understand and appreciate the biology behind the algorithms.

Professor Kevin Skadron has my gratitude for sparking my interest in parallel algorithms. His computer architecture class gave me appreciation for a topic that I was sure to otherwise avoid. Kevin was gracious enough to continue advising and support me well after his course ended and through a difficult series of paper rejections, revisions, and finally acceptance.

I am grateful to all of the current and former members of the Hall and Quinlan labs: Colby Chiang, Royden Clark, Greg Faust, Michael Lindberg, Dr. Ankit Malhotra, Svetelana Shumilina Neil Kindlon, and Dr. Uma Paila. The constant discussions in lab helped many of my mediocre ideas became good ones.

I would also like to thank my mom and sister for their suggestions; my dad, who is not a computer scientist or a biologist, for reading this entire document and making it much better; Ryan Dale for providing a number of scripts that aided in the analysis and interpretation of the results in Chapter 4; Michael Boyer for his help with CUDA; and Johnathan Dorn for his help improving Chapter 5.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

An organism's genome encodes the instructions that guide all development and behavior [1], and can determine an individual's susceptibility or resistance to many classes of diseases. Within a genome, the sequences for all of the different genomic features (e.g., genes, fragile sites, repeat sequences, etc.) are arranged in a linear order along chromosomes. A reference genome provides a common point of comparison by allowing DNA sequences to be mapped to their interval positions within the reference. For example, the BRCA1 gene sequence maps to the interval on chromosome 17 from position 41,196,312 to 41,277,500 in build 37 reference of the human reference genome. Since the completion of first human genome assembly [2], the genomics community has labored to annotate it by identifying the chromosomal intervals that harbor, for example, genes associated with disease, significant conservation across diverse species, or the many functional elements that modulate gene expression. Each such annotation is itself a distinct set of genome intervals, and the majority of discovery in genomics research involves, in some way, comparing the relationships between genomic features. Such comparisons involve screens for interval *intersection*: that is, if feature sets contain intervals that overlap one another, then an underlying biological relationship can often be inferred. As such, an efficient intersection algorithm provides researchers with the ability to identify genomic regions that are common among multiple (or unique to an individual) datasets, and place their experimental results in a broader context.

Considering the recent growth in the amount of high-throughput sequencing data that has become available and the steep drop in the costs associated with sequencing, there is a need for efficient algorithms to process this data. To meet this need, this thesis presents three algorithms that can power the next generation of genomic analysis:

1. Binary Interval Search (BITS), an optimal sequential algorithm and a work-efficient parallel algorithm to count the number of intersections between two interval sets;
2. Split-then-sweep, an efficient divide-and-conquer algorithm to find the intersections among a set of interval sets that also exposes significant amounts of fine-grain parallelism; and
3. LUMPY, a novel algorithm to detect changes to an organism's genome that can integrate many types of high-throughput sequencing evidence and is enabled by interval intersection.

Pair-wise interval set intersection is a fundamental operation that, given two sets of intervals, produces the set of interval pairs in opposing sets that intersect. Interval intersection is the fundamental operation in a broader class of "genome arithmetic" techniques, and as such, it underlies the functionality found in many genome analysis software packages [3, 4, 5, 6]. When comparing large data sets, researchers are often more interested in statistics concerning the intersection than the full list of intersecting intervals. For example, an intersecting set that is larger than expected can indicate that two genomic features are closely related. This type of analysis can be accomplished by counting the number of intersecting intervals, which does not require the fundamentally more difficult problem of full enumeration. Given the need for this type of comparison, and the fact that all of the currently available algorithms are based on enumeration, this thesis introduces the Binary Interval Search (BITS) algorithm, a novel and scalable solution to count the number of intersections between two sets of intervals. BITS uses two binary searches (one each for start and end coordinates) to identify intersecting intervals. The sequential version of BITS is optimal, outperforms existing approaches, and is intrinsically suited to parallel architectures such as the NVIDA CUDA Graphics Processing Unit (GPU) [7]. BITS provides researchers the unique ability to compare thousands of data sets on a single GPU-enabled machine, a task that would otherwise require a cluster with over 100 nodes.

A natural extension to pair-wise intersection is an algorithm to find the N-way intersections among a larger number of interval sets. That is, given a set of N interval sets, determine the set of intervals that intersect across all N sets. Identifying intersections common to many sets is particularly important when attempting to distill biological insights from large, multi-dimensional (e.g., features across tissue types and developmental stages) datasets such as those produced by the ENCODE [8] or Roadmap Epigenomics [9] projects. These projects endeavor to catalog the spectrum of functional elements in the human genome, and in so doing, they have thus far produced thousands of distinct sets of genomic features among many different cell types. Understanding the biological relationships of these functional elements requires a higher-level analysis that considers many sets at once. This thesis introduces a "slice-then-sweep" algorithm as a novel and efficient solution to the N-way intersection problem. While any algorithm that can identify intersections between pairs of interval sets can be extended to find N-way intersections by iterative pair-wise set comparisons, such approaches require a significant amount of overhead to create intermediate results and track intersection provenance. One widely-used algorithm, the linear sweep, has been directly extended to find N-way intersections [10], but may lead to over-processing and has little opportunities for parallelization. In contrast, the slice-then-sweep algorithm attempts to minimize wasted computation and increase the amount of parallelism by creating independent *slices* of the data. The slice step identifies and discards regions (slices) that cannot possibly contain an N-way intersection, and the resulting slices of the data are completely independent and can therefore be efficiently processed by linear sweeps in parallel. While the speedup provided by slicing depends on the number of intervals that can be excluded, experiments have shown that even in the worst case the slice-then-sweep strategy performs as well as a linear sweep. Moreover, by enabling parallelization, slicing provides significant speedups for the analysis of typical large-scale genomics datasets.

Beyond enabling the direct comparison of interval data sets, efficient intersection algorithms also provide a platform for more complex genomic analysis algorithms. This thesis considers the problem of finding changes in chromosome structure between a sample genome and the human reference genome. These changes are collectively known as structural variation (SV), a term that encompasses diverse genomic alterations including deletion, tandem duplication, insertion, inversion, translocation, or complex rearrangement of relatively large (e.g., > 100 base pairs) segments. While SVs are considerably less common than smaller-scale forms of genetic variation, they have a greater functional potential due to their larger size, and they are more likely to alter gene structure or dosage. For example, 95% of patients with chronic myelogenous leukemia have a genome that contains a translocation between chromosomes 9 and 22 that resulted in the oncogenic BCR-ABL gene fusion. Paired-end high throughput sequencing data can provide evidence of SV in the form of several types of signals, including sequences that align in unexpected configurations (read-pair signal) and sequences that contain sub-sequences that align to distant locations (split-read signal). Comprehensive discovery of SVs requires the integration of multiple signals. However, owing to inherent technical challenges, existing SV discovery algorithms either use a single signal in isolation, or at best use two signals sequentially. Such approaches suffer from limited sensitivity when an SV is captured by only a few sequences, as often occurs in low coverage datasets or at somatic variants with low intra-sample frequencies. In particular, sensitive detection of low frequency somatic SVs in heterogeneous tumor samples is an unsolved problem with major clinical implications. To address this problem, this thesis presents LUMPY, a novel and extremely flexible probabilistic SV discovery framework capable of integrating any number of SV detection signals including those generated from read alignments or prior evidence jointly from multiple samples. LUMPY enables straightforward signal integration by mapping each SV signal to a common abstract representation in the form of paired probability distributions, and then performs SV prediction operations at this higher level. These prediction operations are made possible by an efficient data structure that enables interval insertion, deletion, and intersection. This approach allows for simple and natural signal integration, inherently produces a probabilistic measure of breakpoint position, and can be easily extended to new signals as sequencing technologies and SV detection strategies evolve.

The compilation of this work represents a significant contribution to the general field of computational biology and bioinformatics and specifically to the field of SV detection. The BITS and the slice-then-sweep N-way intersection algorithms provide researchers with new comparison operations that can support either direct data set comparison, or enable higher-level analysis. The LUMPY algorithm both demonstrates the utility of an efficient interval intersection platform, and advances the state-of-the-art in SV detection.

The rest of this thesis is organized as follows. Chapter 2 covers the basic concepts and notation regarding interval intersection, genomic analysis, and SV. Chapter 3 is a review of related work in the interval intersection and SV fields. The BITS algorithm is given in Chapter 4, the N-way slice-then-sweep algorithm in Chapter 5, and the LUMPY algorithm in Chapter 6. Finally, conclusions and future work is proposed in Chapter 7.

Chapter 2

Background

2.1 Interval Intersection

An *interval* $a = \langle a.start, a.end \rangle$ is a continuous set of values of between start and end locations (e.g., a gene), and an *interval set* $A = \{a_1, \ldots, a_N\}$ is a collection of intervals (e.g., all known genes). Two intervals a and b *intersect* when $a.start \leq b.end$ and $a.end \geq b.start$. For convenience, let $a = b$ if a and b intersect (Figure 2.1a, 2.1b), $a < b$ if a ends before b starts ($a.end < b.start$) (Figure 2.1c), $a > b$ if a starts after b ends ($a.start > b.end$) (Figure 2.1d), $a \leq b$ if a intersects b and starts before b (Figure 2.1a), and $a \geq b$ if a intersect b and ends after b (Figure 2.1b). It is possible for interval a_i to contain a_j if $a_i \leq a_j$ and $a_i \geq a_j$ (Figure 2.2). Let $a \equiv b$ only if a and b refer to the same interval in the same set.

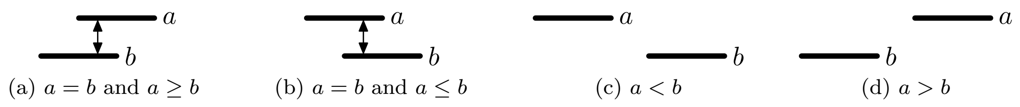


Figure 2.1: Interval equality and inequality relationships.

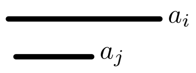
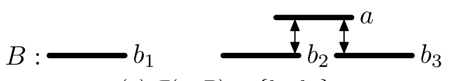
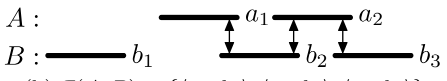


Figure 2.2: Interval a_i contains interval a_j.

The intersection of an interval a and interval set B is the set of intervals: $\mathcal{I}(a, B) = \{b_i | b_i \in B, a = b_i\}$ (Figure 2.3a), and the intersection of two interval sets A and B is the set of interval pairs: $\mathcal{I}(A, B) = \{a_i \times \mathcal{I}(a_i, B) | a_i \in A\}$ (Figure 2.3b). Chapter 4 describes an algorithm that counts the number of intersections between two interval sets.



(a) $\mathcal{I}(a, B) = \{b_2, b_3\}$



(b) $\mathcal{I}(A, B) = \{\langle a_1, b_2\rangle, \langle a_2, b_2\rangle, \langle a_2, b_3\rangle\}$

Figure 2.3: Intersections between intervals and intervals sets.

The intersection among a set of intervals sets $\mathbf{S} = \{S_1, \ldots, S_N\}$, where $S_i = \{s_{i,a}, \ldots, s_{i,b}\}$, (the N-way intersection, Figure 2.4) is a set of interval sets

$$\mathcal{I}(\mathbf{S}) = \left\{ s_{1,j}, s_{2,k}, \ldots, s_{N,l} \;\middle|\; \begin{array}{l} s_{a,b} \in S_a, \\ s_{a,b} = s_{c,d} \forall s_{a,b}, s_{c,d} \in \{s_{1,j}, s_{2,k}, \ldots, s_{N,l}\} \end{array} \right\}$$

Where each set contains exactly one interval from each $S_i \in \mathbf{S}$, and all of the intervals within a set intersect. In the N-way intersection problem, it is assumed that sets are free of contained intervals. This assumption simplifies the problem by eliminating redundant intersections that are all based on the on a common region. Chapter 5 introduces an algorithm that can reduce computation time and allow for parallel execution by partitioning the sets into independent *slices*.

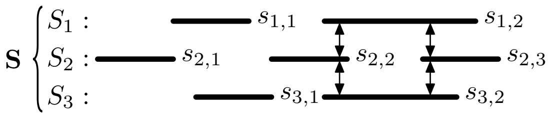


Figure 2.4: The N-way intersection $\mathcal{I}(\mathbf{S}) = \big\{\{s_{1,2}, s_{2,2}, s_{3,3}\}, \{s_{1,2}, s_{2,3}, s_{3,3}\}\big\}$

2.2 Genomic Analysis

Genome assembly is the process of reconstructing the nucleotide sequences that make up the chromosomes in an organism's genome. The current trend in genomic analysis is to define a *reference genome* for each species that is the consensus sequence of several individuals. A reference genome provides a common point of comparison for different types of analysis through a process called *alignment*. Given some DNA sequence x, alignment to the reference $R(x) = \langle c, o, s, e\rangle$ maps x to its relative *genomic location* among the full set of sequences in the reference. A genomic location is defined by a chromosome c, an orientation o (DNA is double stranded, and orientation refers to either the "+" or "−" strand), a start position s, and an end position e. For example, considering the human reference genome and sequence $x = \text{ACGGGGTCTCGAAAAAAGGA}$, $R(x) = \langle 17, -, 41280732, 41280751\rangle$ (Figure 2.5). Depending on the complexity of the genome, the completeness of its reference assembly, and the genetic divergence of the sample, it is possible for a sequence to map to zero locations or many locations.

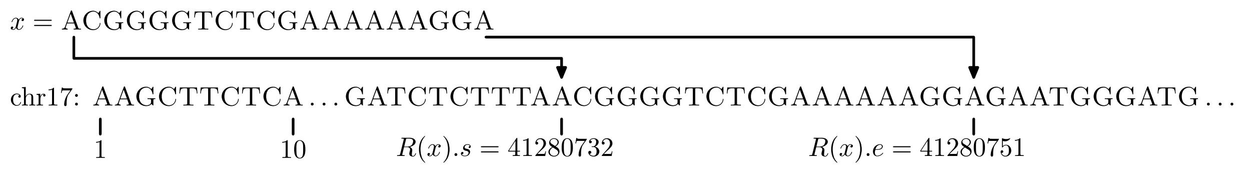

x = ACGGGGTCTCGAAAAAAGGA
chr17: AAGCTTCTCA . . . GATCTCTTTAACGGGGTCTCGAAAAAAGGAGAATGGGATG . . .
1
10
R(x).s = 41280732
R(x).e = 41280751

Figure 2.5: The DNA fragment x aligns to chromosome (chr) 17 of the reference between positions 41,280,732 and 41,280,751.

Paired-end high throughput sequencing is the most widely used genomic sequencing technology. The process involves fragmenting genomic DNA into roughly uniformly-sized fragments, and sequencing both ends of each fragment to produce paired-end reads $\langle x, y\rangle$, which are referred to as *read-pairs* (Figure 2.6). When both x and y align to a single location in the reference and both ends align to the same chromosome ($R(x).c = R(y).c$), it is assumed that x aligns before y ($R(x).s < R(x).e < R(y).s < R(y).e$). If $R(x).c \neq R(y).c$, then $R(x).c < R(y).c$. While in practice it is not possible to know the position of read x in the sample genome (in the absence of whole-genome assembly), it is useful to refer to $S(x) = \langle c, o, s, e\rangle$ as the alignment of x with respect to the originating samples genome.

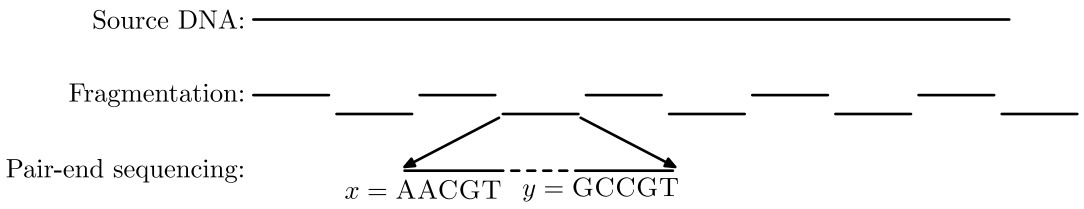


Figure 2.6: Source DNA is sheered into similarly sized fragments, and each fragment undergoes pair-end sequenced that produces the sequences x and y. In pair-end sequencing, the regions between x and y (dashed line) are unsequenced.

In some cases, an end of a read-pair that does not align to a single location contains subsequences that align to distant locations within the reference. These *split-read alignments* are often evidence of differences between a sample and the reference. A split-read alignment is a single DNA fragment X (e.g., one end of a paired-end read) that does not contiguously align to the reference genome (Figure 2.7). Instead, X contains a set of two or more substrings $x_i \ldots x_j$ ($X = x_1 x_2 \ldots x_n$), where each substring aligns to the reference $R(x_i) = \langle c, o, s, e\rangle$, and adjacent substrings align to non-adjacent locations in the reference genome $R(x_i).e \neq R(x_{i+1}).s + 1$ for $1 \leq i \leq n - 1$. A single split-read alignment maps to a set of adjacent split-read sequence pairs ($\langle x_1, x_2\rangle, \langle x_2, x_3\rangle, \ldots, \langle x_n - 1, x_n\rangle$), and each pair $\langle x_i, x_i + 1\rangle$ is considered individually.

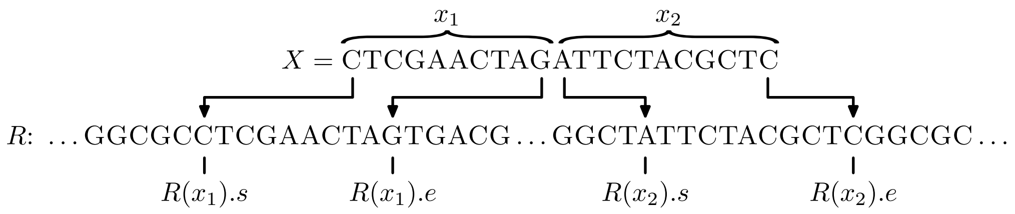

x_1
x_2
$X =$ CTCGAACTAGATTCTACGCTC
R: ...GGCGCCTCGAACTAGTGACG...GGCTATTCTACGCTCGGCGC...
$R(x_1).s$
$R(x_1).e$
$R(x_2).s$
$R(x_2).e$

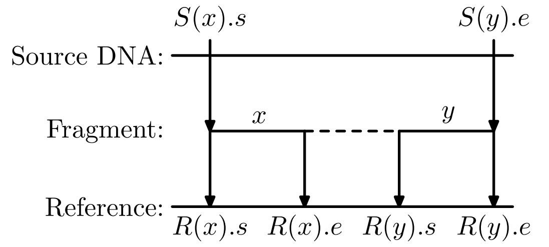
Figure 2.7: The sequence X produces a split-read alignment when its subsequences x_1 and x_2 align to distant regions of the reference. In this case $R(x_2).s \gg R(x_1).e$.

2.3 Structural Variation

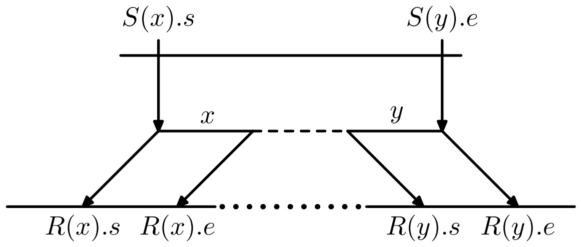
One common type of genomic analysis is to identify large differences (> 100 bp) between a sample genome and the reference. These differences, referred to as structural variation (SV), have the potential to alter gene structure or dosage. Paired-end high-throughput sequencing data can provide evidence of SV in the form of several types of signals, including sequences that align in unexpected configurations (read-pair signal) and sequences contain subsequences that align to distant locations (split-read signal). Most SV detection algorithms are based on a single signal, or consider signals one at a time. Chapter 6 introduces an algorithm that pools all available signals.

The expected alignment for a DNA fragment with paired-end reads $\langle x, y \rangle$, is for the two ends to be from the same chromosome ($R(x).c = R(y).c$), the first end to be on the + strand ($R(x).o = +$), the second end to be on the − strand ($R(y).o = -$), and for the outer distances between the ends to match the fragment size ($R(y).e - R(x).s = S(y).e - S(x).s$) (Figure 2.8a). Any other configuration is a possible SV signal. For example, a read-pair that spans a region of the sample genome that is missing a stretch of DNA (a deletion with respect to the reference) will result in a pair with ends that align further from each other than expected ($R(y).e - R(x).s \gg S(y).e - S(x).s$) (Figure 2.8b), and a sample in which two chromosomes have been spliced together (a translocation) will result in a pair with mismatched chromosomes ($R(x).c \neq R(y).c$). Other configurations, such as matching orientations, are signals for different types of SV (duplications, insertions, and inversions).

Since the expected alignment for a single fragment X is for the the full fragment to align contiguously ($R(x_i).e = R(x_{i+1}).s + 1$), the existence of a split-read is a possible SV signal. For example, if a X spans a deletion, then a large gap with exist between two of the alignments within X ($R(x_i).e \ll R(x_{i+1}).s$), and if X spans a translocation then the alignments will not have matching chromosomes ($R(x_i).c \neq R(x_{i+1}).c$).



(a) $R(y).e - R(x).s = S(y).e - S(x).s$ is the expected alignment.



(b) $R(y).e - R(x).s \gg S(y).e - S(x).s$ indicates a region of DNA (dotted line) in reference has been deleted in the source.

Figure 2.8: Unexpected pair-end alignments can indicate that a region of the source DNA differs from the reference.

Chapter 3

Related Work

3.0.1 Interval Intersection

Interval intersection has many applications in genomics, and several algorithms have been developed that, in general, are either based on trees [3, 11], or linear sweeps of pre-sorted intervals [12].

The UCSC genome browser introduced a widely-used scheme based on R-trees. This approach partitions intervals from one dataset into hierarchical “bins.” Intervals from a second dataset are then compared to matching bins (not the entire dataset) to narrow the search for intersections to a focused portion of the genome. While this approach is used by the UCSC Genome Browser, BEDTools [13], and SAMTOOLS [5], the algorithm is inefficient for counting intersections since all intervals in each candidate bin must be *enumerated* in order to count the intersections. Since the number of intersections is at most quadratic, any enumeration-based algorithm is $O(N^2)$.

Moreover, these existing approaches are poor candidates for parallelization. Thread divergence can be a significant problem for hierarchical binning methods. If intervals are not uniformly distributed (e.g., exome sequencing or RNA-seq), then a small number of bins will contain many intervals while most other bins will be empty. Consequently, threads searching full bins will take substantially longer than threads searching empty bins.

Recent versions of BEDTools and BEDOPS [14] conduct a linear “sweep” through pre-sorted datasets while maintaining an auxiliary data structure to track intersections as they are encountered. While the complexity of such sequential sweep algorithms is theoretically optimal, the amount of parallelism that exists is limited, and some overhead is required to guarantee correctness. Any linear sweep algorithm must maintain the “sweep invariant” [15], which states that all segment starts, ends, and intersections behind the sweep must be known. A parallel sweep algorithm must either partition the input space such that each section can be swept in parallel without violating the invariant, or threads must communicate about intervals that span partitions. In the first case parallelism is limited to the number of partitions that can be created, and threads can diverge when the number of intervals in each partition is unbalanced. In the second case, the communication overhead between threads prevents work efficiency and can have significant performance implications. In BITS, the amount of parallelism depends only on the number of intervals and not the distribution of intervals within the input space, and there is no communication between threads.

Any pair-wise algorithm can find N-way intersections by iteratively comparing sets. However, this approach requires a significant amount of overhead to create intermediate results and track intersection pedigree. One widely-used algorithm, the linear sweep, can be directly extended to find N-way intersections [10], but may lead to over-processing and has little opportunities for parallelization. To address these problems, the slice-then-sweep algorithm minimizes over-processing and increase parallelism by creating independent *slices* of the data. Slices that cannot contain an N-way intersection is discarded, and the remaining subsets of the data are processed in parallel.

3.0.2 Structural Variation

Given the importance of SV detection, a number of methods have been proposed. The basic strategy for all methods is to cluster SV signals that have similar breakpoint locations. Most of the available methods utilize one SV signal, typically *aberrant* read-pairs that aligned to the reference genome in an unexpected configuration.

HYDRA [13] groups aberrant read-pairs that span the same genomic interval. Groups are then iteratively pruned to maximize the number of mutually supported read-pairs. BreakDancerMax [16] identifies large SV (> 100 bp) by scanning the genome for regions with more aberrant read-pair ends than expected, and makes a prediction for any two regions connected by at least two read-pairs. BreakDancerMini [16] identifies small SV ($10 - 100$ bp) by using a sliding window to compare the distance between the ends of normal read-pairs in the window to the expected distance. GASV [17] utilizes knowledge of the fragment size to map aberrant read-pairs that to a polygon in 2D space $G \times G$ (G is a concatenation of chromosomes in the reference genome) that contains the breakpoint. Read-pairs that span the same breakpoint will map to intersecting polygons. After all of the polygons have been created, a plane sweep algorithm is used to find polygons that intersect. Any region that is common to a minimum number of polygons reported as a breakpoint.

Newer methods integrate a second SV signal to refine predictions made from aberrant read-pairs. GASVPro [18] is an extension of GASV that uses read-depth to refine reported breakpoints. Read-depth refers to the number of correctly aligned read-pairs that cover a given region, and the GASVPro authors observed that the number of read-pairs that align correctly should decrease in the region surrounding a true breakpoint. Based on this, a probabilistic model that compares the actual coverage to the expected is used to assign a likelihood to breakpoints reported by the GASV method. DELLY [19] refines predictions made from read-pairs by attempting to reconstruct the breakpoint sequence. A breakpoint joins non-adjacent reference regions to create a novel junction sequence. When a junction sequence is captured by one end of a read-pair, the end will, at best, only partially align to the reference. DELLY scans each region identified by aberrant read-pairs for these partially aligning ends. The ends are processed by an assembly algorithm in an attempt to reconstruct the junction sequence. The objective is for the read-pair prediction and the breakpoint assembly to corroborate the SV location.

Chapter 4

Binary Interval Search (BITS): A Scalable Algorithm for Counting Interval Intersections

4.1 Introduction

The Binary Interval Search (BITS) algorithm is a novel and scalable solution to the fundamental problem of counting the number of intersections between two sets of genomic intervals. BITS uses two binary searches (one each for start and end coordinates) to identify intersecting intervals. As such, the algorithm executes in $\Theta(M \log M)$ time, where M is the number of intervals, which can be shown to be optimal by a straight-forward reduction to element uniqueness (known to be $\Theta(M \log M)$ [20]). In contrast, counting intersections by enumeration is less efficient, as enumerating intervals requires time $\Theta(W + M \log M)$, where W is the number of intersections which can be quadratic to the input size. The sequential version of BITS outperforms existing approaches, and BITS is intrinsically suited to parallel architectures. The parallel version performs the same amount of work as the sequential version (i.e., there is no overhead) which means the algorithm is work-efficient, and because each parallel thread performs equivalent work, BITS has little thread divergence. Although thread divergence degrades performance on any architecture (finished threads must wait for over-burdened threads to complete), the impact is particularity acute for GPUs where threads share a program counter and any divergent instruction must be executed on every thread.

4.2 Methods

A seemingly facile method for finding the intersection of A and B would be to treat one set, A, as a "query" set, and the other, B, as a "database". If all of the intervals in the database were sorted by their starting coordinates, it would seem that binary searches could be used for each query to identify all intersecting database intervals.

However, this apparently straight-forward searching algorithm is complicated by a subtle detail. If the intervals in B are sorted by their starting positions, then a binary search of B for the query interval end position $a_i.end$ will return the interval $b_j \in B$, where b_j is the last interval in B that starts before interval a_i ends (e.g, interval e in Figure 4.1A). This would seem to imply that if b_j does not intersect a_i, then no intervals in B intersect a_i, and if b_j does intersect a_i, then other intersecting intervals in B could be found by scanning the intervals starting before b_j in decreasing order, stopping at the first interval that does not intersect a_i. However, this technique is complicated by the possibility of intervals that are wholly *contained* inside other intervals (e.g., interval c in Figure 4.1B).

An interval $b_j \in B$ is "contained" if there exists an interval $b_k \in B$ where $b_k.start \leq b_j.start$ and $b_j.end \leq b_k.end$. Considering such intervals, if the interval found in the previous binary search b_j does not intersect the query interval a_i, it cannot be concluded that no interval in B intersects a_i, because there may exist an interval $b_{j-x} \in B$ where $b_{j-x}.end > a_i.start$. Furthermore, if b_j does intersect a_i, then the subsequent scan for other intersecting intervals cannot stop at the first interval that does not intersect a_i; it is possible that some earlier containing interval intersects a_i. Therefore, the scan is forced to continue until it reaches the beginning of the list. As contained intervals are typical in genomic datasets, a naive binary search solution is inviable.

4.2.1 Binary Interval Search (BITS) Algorithm

The Binary Interval Search (BITS) algorithm uses two binary searches to identify interval intersections while avoiding the aforementioned complexities caused by contained intervals. The key observation underlying BITS is that the *size* of the intersection between two sets can be determined without enumerating each intersection. For each interval in the query set, two binary searches are performed to determine the number of intervals in the database that intersect the query interval. Each pair of searches is independent of all others, and thus all searches can be performed in parallel.

Existing methods define the intersection set based on *inclusion*: that is, the set of intervals in the interval database B that end after the query interval a_i begins, and which begin before a_i ends. However, contained intervals make it difficult to find this set directly with a single binary search.

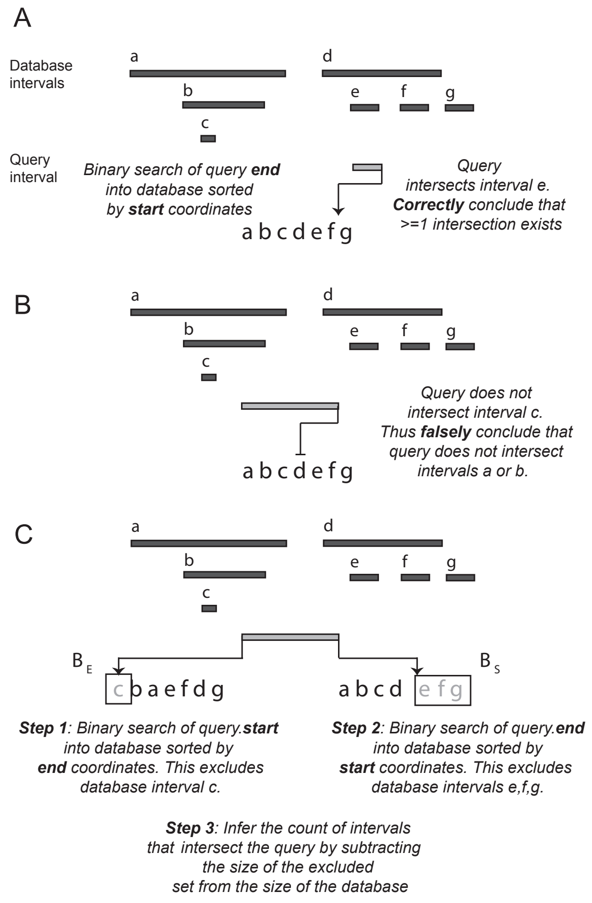

A
Database intervals
a
b
c
d
e
f
g
Query interval
Binary search of query **end** into database sorted by **start** coordinates
a b c d e f g
Query intersects interval e. **Correctly** conclude that >=1 intersection exists
B
Query does not intersect interval c. Thus **falsely** conclude that query does not intersect intervals a or b.
C
B E
c b a e f d g
B S
a b c d e f g
Step 1: Binary search of query.**start** into database sorted by **end** coordinates. This excludes database interval c.
Step 2: Binary search of query.**end** into database sorted by **start** coordinates. This excludes database intervals e,f,g.
Step 3: Infer the count of intervals that intersect the query by subtracting the size of the excluded set from the size of the database

Figure 4.1: Comparing a naive binary search for interval intersection to the BITS approach. **A**. Binary searches of intervals sorted by start coordinate will occasionally identify overlapping intervals. However, contained intervals prevent knowing how far one must scan the database to identify all intersections. **B**. Contained intervals also cause single binary searches to falsely conclude that no intersections exist for a given query interval. **C**. To overcome these limitations, BITS uses two binary searches of the database intervals: one into a sorted list of end coordinates and the other into a sorted list of start coordinates. Each search excludes database intervals that *do not* intersect the query, leaving solely the intervals that *must* intersect the query.

BITS is based on a different, but equivalent, definition of interval intersection based on *exclusion*: that is, by identifying the set of intervals in B that *do not* intersect a_i, the number of intervals that *must* intersect a_i can be inferred. Formally, the set of intervals $\mathcal{I}(B, a_i) \in B$ that intersect query interval $a_i \in A$ are the intervals in B that are neither in the set of intervals ending before (“left of”, set $\mathcal{L}$ below) a_i begins, nor in the set of intervals starting after (“right of”, set $\mathcal{R}$ below) a_i ends. That is:

$$\mathcal{L}(B, a_i) = \{b \in B | b.end < a_i.start\}$$

$$\mathcal{R}(B, a_i) = \{b \in B | b.start > a_i.end\}$$

$$\mathcal{I}(B, a_i) = B \setminus (\mathcal{L}(B, a_i) \cup \mathcal{R}(B, a_i))$$

Finding the intervals in $\mathcal{I}(a_i, B)$ for each $a_i \in A$ by taking the difference of B and the union of $\mathcal{L}(B, a_i)$ and $\mathcal{R}(B, a_i)$ is not efficient.

However, the size of $\mathcal{L}(B, a_i)$ and the size $\mathcal{R}(B, a_i)$ can be easily found and then used to then *infer* the size of $\mathcal{I}(B, a_i)$. From the size of $\mathcal{I}(B, a_i)$, the decision problem, the counting problem, and the per-interval counting problems can be directly answered. The size of $\mathcal{I}(B, a_i)$ also serves as the termination condition for enumerating intersections that was missing in the naive binary search solution.

Algorithm 1: Single interval intersection counter

```
Input: Sorted interval starts and ends $B_S$ and $B_E$, query interval $a$
Output: Number of intervals $c$ intersecting $a$
Function ICount($B_S, B_E, a$) begin
    $first \leftarrow$ BinarySearch($B_S, a.end$)
    $last \leftarrow$ BinarySearch($B_E, a.start$)
    $c \leftarrow first - last$                    /* $= |B| - (last + (|B| - first))$ */
    return $c$
```

The BITS algorithm is based upon one fundamental function, $\textsc{ICount}(B_S, B_E, a_i) = |\mathcal{I}(B, a_i)|$ (Algorithm 1), which determines the number of intervals in the database B that intersect query interval a_i. As shown in Figure 4.1C, the database B is split into two integer lists $B_S = [b_1.start, b_2.start, \ldots, b_M.start]$ and $B_E = [b_1.end, b_2.end, \ldots, b_M.end]$, which are each sorted numerically in ascending order. Next, two binary searches are performed, $last = \textsc{BinarySearch}(B_E, a_i.start)$ and $first = \textsc{BinarySearch}(B_S, a_i.end)$. Since B_E is a sorted list of each interval end coordinate in B, the elements with indices less than or equal to $last$ in B_E correspond to the set of intervals in B that end *before* a_i starts (i.e., to the "left" of a_i). Similarly, the elements with indices greater than or equal to $first$ in B_S correspond to the set of intervals in B that start *after* a_i ends (i.e., to the "right" of a_i). These two values are used to directly infer the size of the intersection set $\mathcal{I}(B, a_i)$ (i.e., the *count* of intersections in B for a_i):

$$\begin{aligned} |B| - first &= |\mathcal{R}(B, a_i)| \\ last &= |\mathcal{L}(B, a_i)| \\ |B| - (last + (|B| - first)) &= |\mathcal{I}(B, a_i)| \end{aligned}$$

The BITS solution to the counting problem. Since BITS operates on arrays of generic intervals ($\langle start, end \rangle$), and input files are typically chromosomal intervals ($\langle chrom, start, end \rangle$), the intervals in each dataset are first projected down to a one-dimensional generic interval. This is a straight forward process that adds an offset associated with the size of each chromosome to the start and end of each interval. Once the inputs are projected to interval arrays A and B, the Counter (Algorithm 2) sets the accumulator variable c to zero; then for each $a_i \in A$, accumulates $c = c + \textsc{ICount}(B_S, B_E, a_i)$. The total count c is returned.

Algorithm 2: Interval intersection counter

```
Input: Database interval array $B$ and query interval array $A$
Output: Number of intersections $c$ between $A$ and $B$
Function Counter($A, B$) begin
    $B_S \leftarrow [b_1.start, \ldots, b_{|B|}.start]$
    $B_E \leftarrow [b_1.end, \ldots, b_{|B|}.end]$
    Sort($B_S$)
    Sort($B_E$)
    $c \leftarrow 0$
    for $i \leftarrow 1$ to $|A|$ do
        $c \leftarrow c +$ ICount($B_S, B_E, A[i]$)
    return $c$
```

4.2.2 Time Complexity Analysis

The time complexity of BITS is $O((|A|+|B|)\log|B|)$, which can be shown to be optimal by a straight-forward reduction to element uniqueness (known to be $\Theta(M \log M)$ [20]). To compute ICount(B_S, B_E, a_i) for each a_i in A, the interval set B is first split into two sorted integer lists B_S and B_E, which requires $O(|B|\log|B|)$ time. Next, each instance of ICount(B_S, B_E, a_i) searches both B_S and B_E, which consumes $O(|A|\log|B|)$ time. For the counting problems, combining the results of all ICount(B_S, B_E, a_i) instances into a final result can be accomplished in $O(|A|)$ time.

4.2.3 Parallel BITS

Performing a single operation independently on many different inputs is a classic parallelization scenario. When based on the subroutine ICount(B_S, B_E, a), which is independent of all ICount(B_S, B_E, x) for intervals x in the query set where $a \neq x$, counting interval intersections is a *pleasingly parallelizable* problem that easily maps to a number of parallel architectures.

NVIDIA's CUDA is a single instruction multiple data (SIMD) architecture that provides a general interface to a large number of parallel GPUs. The GPU is organized into multiple SIMD processing units, and the processors within a unit operate in lock-step. The BITS algorithm is especially well suited for the this architecture for a number of reasons. First, CUDA is optimized to handle large numbers of threads. By assigning each thread one instance of ICount(B_S, B_E, a), the number of threads will be proportional to the input size. CUDA threads also execute in lock-step and any divergence between threads will cause reduced thread utilization. While there is some divergence in the depth of each binary search performed by ICount(B_S, B_E, a), it has an upper bound of $O(log|B|)$. Outside of this divergence ICount(B_S, B_E, a) is a classic SIMD operation [21]. Finally, the only data structure required for this algorithm is a sorted array, and thanks to years of research in this area, current GPU sorting algorithms can sort billions of integers within seconds [22, 23].

4.3 Results

4.3.1 Comparing BITS to Extant Sequential Approaches

A sequential version of the BITS algorithm ("BITS-SEQ") was implemented as a stand-alone C++ utility. The performance of this implementation was assessed relative to BEDTOOLS `intersect` and UCSC Genome Browser's ("UCSC") [3] `bedIntersect` utilitiesfor *counting* the total number of observed intersections between sets of intervals of varying sizes (Figure 4.2). The comparisons presented are based on sequence alignments for the CEU individual NA12878 by the 1000 Genomes Project [24], as well as RefSeq exons. All run-times were measured on a 2.66 GHz quad-core Intel Xeon X555 CPU with 8 MB of cache running Ubuntu Linux version 4.4.3 (kernel version 2.6.32-34). Run-times for CUDA were measured on an NVIDIA Tesla C2050 GPU with 448 1.15 GHz cores, 3 GB of global memory, CUDA driver version 4246739, and CUDA runtime version 4.0. The source code was compiled using gcc version 4.4.3, and NVIDIA CUDA compilation tools release 4.0, V0.2.1221. In each case, run-times measure the performance of the intersection algorithm and thus do not include disk reading, writing, or the time required to initialize the GPU.

Owing to the different data structures used by each algorithm, the relative performance of each approach may depend on the genomic distribution of intervals within the sets. As discussed previously, tree-based solutions that place intervals into hierarchical bins may perform poorly when intervals are unevenly distributed among the bins. The impact of differing interval distributions was tested by randomly sampling 1 and 10 million alignment intervals from both whole-genome and exome-capture datasets for NA12878. Each algorithm was evaluated considering three different interval intersection scenarios. First, the intervals from *different* distributions were tested by comparing the intersection between exome-capture alignments and whole-genome alignments. Since each set has a large number of intervals and a different genomic distribution, the expected number of intersections is a small (relative to the total set size). Next, a *uniform* distribution was tested by counting intersections between Refseq exons and whole-genome sequencing alignments. Here each interval set is, for the most part, evenly distributed throughout the genome; thus, each exon is expected to intersect roughly the same number of sequencing intervals, and a large number of sequencing intervals will not intersect an exon. Lastly, a *biased* intersection distribution was tested by counting intersections between exons and exome-capture alignments. By design, exome sequencing experiments intentionally focus collected DNA sequences to the coding exons. Thus, the vast majority of sequence intervals will align in exonic regions. In contrast to the previous scenario, nearly every exon interval will have a large number of sequence interval intersections, and nearly all sequencing intervals will intersect an exon.

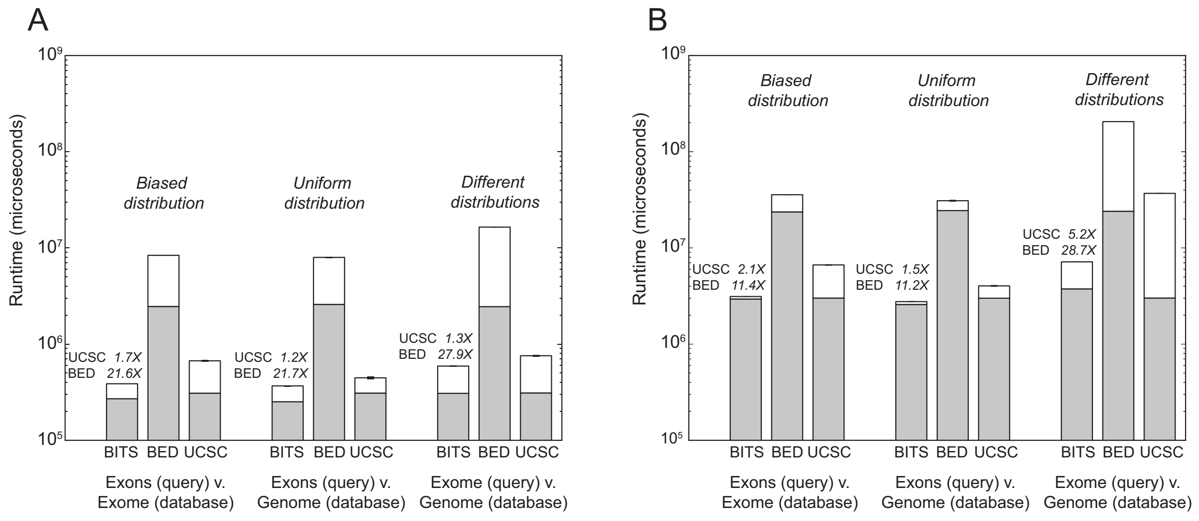


Figure 4.2: Run times for counting intersections with BITS, BEDTools, and UCSC "Kent source". **A**. Run times for databases of 1 million alignment intervals from each interval distribution. **B**. Run times for databases of 10 million alignment intervals from each interval distribution. Bars reflect the mean run time from five independent experiments and error bars describe the standard deviation. Gray bars reflect the run time consumed by data structure construction, while white bars are the time spent counting intersections. Above each BITS execution time the speed increase is given relative to BEDTools and "Kent source", respectively. "Exons" represents 400,351 RefSeq exons (autosomal and X, Y) for the human genome (Build 37). BED = BEDTools; UCSC = "Kent source".

BITS excels at counting intersections.

In all three interval distribution scenarios, the sequential version of BITS had superior runtime performance for counting intersections. BITS was between 11.2 and 27.9 times faster than BEDTools and between 1.2 and 5.2 times faster than UCSC (Figure 4.2). This behavior is expected; whereas the BEDTools and UCSC tree-based algorithms must *enumerate* intersections to derive the *count*, BITS *infers* the intersection count by exclusion without enumeration.

BITS excels at large intersections and biased distributions.

The relative performance gains of the BITS approach are enhanced for very large datasets (Figure 4.2B). Since tree-based methods have a fixed number of bins, and searches require a linear scan of each associated bin, the number of intervals searched grows linearly with respect to the input size. In the worst-case where all intervals are in a single bin, a search would scan the entire input set. In contrast, BITS employs binary searches so the number of operations is proportional to log of the input size, regardless of the input distribution.

Similarly, exome-capture experiments yield biased distributions of intervals among the UCSC bins. Consequently, most bins in tree-based methods will contain no intervals, while a small fraction contain many intervals. When the query intervals have the same bias, the overhead of the UCSC algorithm is more onerous, as a small number of bins are queried and each queried bin contains many intersecting intervals that must be enumerated in order to count overlaps. As the BITS algorithm is agnostic to the interval distributions, it will outperform the UCSC algorithm (Figure 4.2A, 4.22B) for common genomic analyses such as ChIP-seq and RNA-seq, especially given the massive size of these datasets.

4.3.2 Applications for Monte Carlo Simulations

Identifying statistically significant relationships between sets of genome intervals is fundamental to genomic research. However, owing to complex evolutionary history, different classes of genomic features have distinct genomic distributions, and as such, testing for significance can be challenging. One widely-used, yet computationally intensive alternative solution is the use of MC simulations that compare *observed* interval relationships to an *expectation* based on randomization. All aspects of the BITS algorithm are particularly well suited for MC simulations measuring relationships between interval sets. As described, all intersection algorithms begin detecting intersections between two interval sets by setting up their underlying data structures (e.g., trees or arrays). The BITS setup process involves mapping each interval from the two-dimensional chromosomal interval space (i.e., chromosome and start/end coordinates) to a one dimensional integer interval space (i.e., start/end coordinates ranging from 1 to the total genome size). Once the intervals are mapped, arrays are sorted by either start or end coordinates. In contrast, the UCSC setup places each interval into a hash table. As shown in Figure 4.2, data structure setup is a significant portion of the total runtime for all approaches.

However, in the case of many MC simulation rounds where a uniformly distributed random interval set is generated and placed into the associated data structure, the setup step is faster in BITS, whereas the setup time remains constant in each simulation round for UCSC. For BITS, the mapping step is skipped in all but the first round and in each simulation round only an array of random starts must be generated. The result is a 6x speedup for MC rounds over the cost of the initial intersection setup. For UCSC, both the chromosome and the interval start position must be generated and then placed into the hash table with no change in execution time.

This speedup in BITS is extended on parallel platforms where the independence of each intersection is combined with efficient parallel random number generation algorithms [25] and parallel sorting algorithms [22, 23]. MC simulations have obvious task parallelism since each round is independent. BITS running on CUDA ("BITS-CUDA") goes a step further and exposes fine-grain parallelism in both the setup step, with parallel random number generation and parallel sorting, and the intersection step where hundreds of intersections execute in parallel. The improvement is modest for a single intersection (only parallel sorting can be applied to the setup step) where BITS-CUDA is 4x faster than sequential BITS and 40x faster than sequential UCSC. However, as the number of MC rounds grows performance improves dramatically. At 10,000 MC rounds and 1e7 intervals, BITS-CUDA is 267x faster than sequential BITS and 3,414x faster than sequential UCSC. An improvement of this scale allows MC analyses for thousands of experiments (e.g., 25,281 pairwise comparisons in Section 4.3.3).

BITS has better performance than UCSC for MC simulations that measure the significance of the overlaps between interval sets in Table 4.1. As both the number of MC rounds and the size of the dataset grows, the speedup of both sequential BITS and BITS-CUDA increases over UCSC. For the largest comparison (1e7 intervals and 10,000 iterations), BITS-SEQ is 12x faster than UCSC, and BITS-CUDA is 267x faster than BITS-SEQ and 3,414x faster than sequential UCSC.

Table 4.1: Runtime (seconds) comparison for MC simulations. Timings in italics were extrapolated owing to long run times.

		Number of MC iterations			
Size	Tool	1	100	1000	10000
1e5	BITS-CUDA	0.73	1	4	28
	BITS-SEQ	0.41	7	68	680
	UCSC	0.17	14	138	1,381
1e6	BITS-CUDA	2	3	1	103
	BITS-SEQ	5	120	*1,200*	*12,000*
	UCSC	6	878	*8,780*	*87,800*
1e7	BITS-CUDA	14	22	97	835
	BITS-SEQ	66	2,235	*22,350*	*223,500*
	UCSC	568	28,508	*285,080*	*2,850,800*

4.3.3 Uncovering Novel Genomic Relationships

The efficiency of BITS for MC applications on GPU architectures provides a scalable platform for identifying novel relationships between large scale genomic datasets. To illustrate BITS-CUDA's potential for large-scale data mining experiments, a screen was performed for significant genomic co-localization among 159 genome annotation tracks using MC simulation. This analysis was based upon functional annotations from the ENCODE project [26] for the GM12878, H1-hESC, and K562 cell lines, including assays for 24 transcriptions factors (often with replicates), 8 histone modifications, open chromatin, and DNA methylation. Diverse genome annotations from the UCSC genome browser (e.g, repeats, genes, and conserved regions) were also included.

BITS-CUDA was used to find the log2 ratio of the observed and expected number of intersections for each of the 25,281 (i.e., 159*159) pairwise dataset relationships using 1e4 MC simulations (Figure 4.3). As expected, this analysis revealed that 1) the genomic locations for the same functional element are largely consistent across replicates and cell types, 2) methylated and semi-methylated regions are similar across cell types, and 3) most functional assays were anti-correlated with genomic repeats (e.g., microsatellites) owing to sequence alignment strategies that exclude repetitive genomic regions. Perhaps not surprisingly, this unbiased screen also revealed intriguing patterns. First, the strong enrichment among all transcription factors (TF) assays suggests that a subset of TF binding sites are shared among all factors. This observation is consistent with previous descriptions of "hot regions" [27]. In addition, there is a significant, specific, and unexplained enrichment among the Six5 TF and segmental duplications.

Pursuing the biology of these relationships is beyond the scope of the current manuscript. However, the ability to efficiently conduct such large-scale screens facilitates novel insights into genome biology. This analysis presented a tremendous computational burden made feasible by the facility with which the BITS algorithm could be applied to GPU architectures. Indeed, each iteration of the MC simulation tested for intersections among 4 billion intervals among the 25 thousand datasets, yielding over 44 trillion comparisons for the entire simulation. Whereas this simulation took 9,069 minutes on a single computer with one GPU card. It would take at least 112 traditional processors to conduct the same analysis using standard approaches such as the UCSC tools or BEDTools.

4.4 Conclusion

BITS is an optimal algorithm for interval intersection that is uniquely suited to scalable computing architectures such as GPUs. This algorithm takes a new approach to counting intersections; unlike existing methods that must enumerate intersections in order to derive a count, BITS uses two binary searches to directly infer the count by excluding intervals that *cannot* intersect one another. By restating the problem this way, BITS provides a speedup of 12x in the sequential case and 3,414x in the parallel case over existing methods. This level of improvement allows researchers to make comparisons between thousands of data sets that require trillions of comparisons on a single machine.

While BITS is a scalable solution for comparing pairs of interval sets, it is not well suited for the discovery of complex biologic relationships that exist across many genomic features. The ability to find intersections across N data sets (the N-way intersection problem) is particularly important given the recent efforts of projects such as ENCODE [8] and Epigenetics Roadmap [9] which are continuously producing large volumes of multi-dimensional data sets. Pair-wise algorithms like BITS have been extended to the N-way intersection problem, but they do not scale due to overhead and a lack of parallelism. In the next chapter the key insight from BITS —that an interval set can be efficiently partitioned into subsets of intervals that begin before and

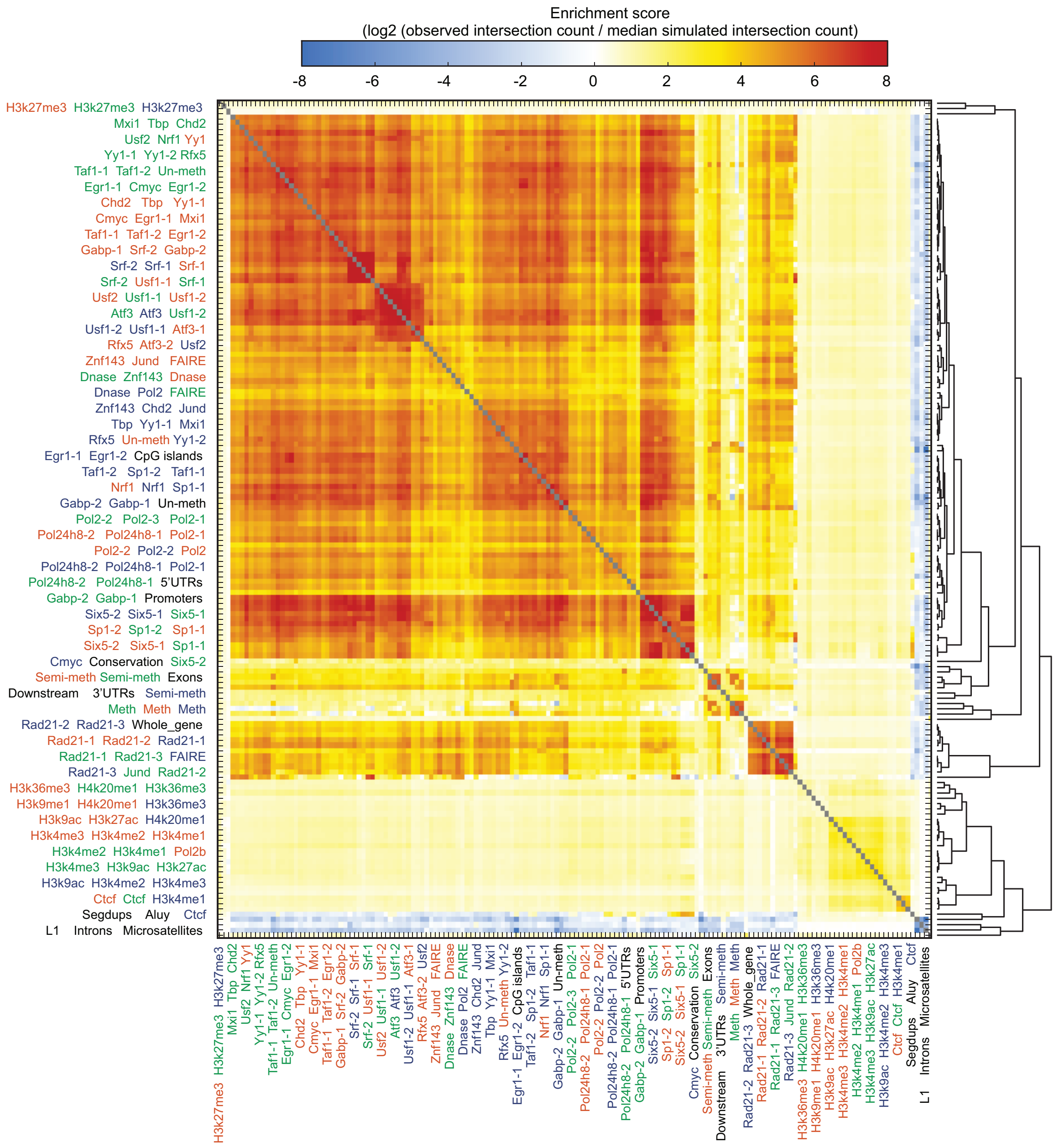


Figure 4.3: BITS-CUDA measurements of spatial correlations among 159 genome features from the ENCODE project and from the UCSC Genome Browser. Each comparison gives an enrichment score reflecting the log2 ratio of the observed count of intersections over the median count of intersections from 10000 MC simulations. Each set of three labels on the X and Y axes correspond to three consecutive rows or columns, respectively. Assays from the GM12878 cell line are in green, H1-hESC in blue, and K562 in red. Annotation tracks from the UCSC Genome Browser are in black.

after a particular interval– is used as the basis for a new algorithm that can reduce that amount of work required to find intersections across many sets and exposes a significant amount of fine-grain parallelism.

Chapter 5

Slice-then-Sweep: A Parallel Algorithm for N-way Interval Set Intersection

5.1 Introduction

Understanding complex biologic relationships between functional elements requires the comparison of many genomic features. For example, proteins that bind to DNA (transcription factors) are known to regulate gene expression, and the presence of certain chromatin markers (histone modifications) has been correlated with gene transcription. Therefore, finding genomic intervals that are common across transcription factor binding sites and several relevant histone modification sites can lead to the discovery of novel gene enhancers. Algorithms that find the intersection between two genomic interval sets have received considerable attention [11, 3, 14, 12, 10, 28], but the ability of these algorithms to consider many interval sets is limited. Interval sets can be intersected iteratively, but there is considerable overhead associated with creating the intermediate results and tracking intersection provenance. The linear sweep algorithm can be directly extended to the N-way intersection problem, but over-processing and a lack of parallelization limit its scalability.

This chapter introduces a novel "slice-then-sweep" algorithm to solve the N-way intersection problem that is based on the same principle used by the BITS algorithm from the previous chapter: interval sets can be efficiently partitioned into subsets of intervals that begin before and after a particular interval. The first step in the algorithm uses this technique to quickly divide the N sets into independent *slices*. Slices that cannot possibly contain an N-way intersection are discarded, and the remaining sets are processed by linear sweeps in parallel. Evolutionary pressure has driven organisms to become as efficient as possible, and most individual genomic features play a role in more than one cellular process. The effect of this diversity is that a comparison of a set of genomic features will typically have many more intervals than N-way intersections, and slicing can significantly reduce that amount of data that is considered by the subsequent sweeps. However, even in the worst case where none of the slices can be discarded, the sequential slice-then-sweepstrategy still performs as well as the standard linear sweep. Furthermore, the amount of parallelism exposed by slicing is independent of the data-reduction and provides significant speedups in all cases.

5.2 Methods

It is clear that any pair-wise interval intersection algorithm can be extended to the N-way problem by considering pairs of sets iteratively; however, the overhead associated with this method limits its practical use. Given a set $\mathbf{S} = \{S_1, \ldots, S_N\}$, the result of $\mathcal{I}(S_1, S_2) = R_{1,2}$ can be input into an intersection with S_3 by mapping the set of pairs in $R_{1,2}$ to a set of intervals that contain the regions common to each pair. That result is then paired with with S_4, and so on. Once all interval sets have been considered, the final set contains intervals that are common to each N-way intersection. From those common intervals, the full N-way intersection can be constructed by either tracking back through each intersection, or using a secondary data structure to maintain a list of intervals common to each result. This extra bookkeeping, the effort required to create the $N-1$ intermediate results, and the fact that regions are being reconsidered many times, adds a considerable amount of overhead.

One widely used algorithm, the linear sweep, can be efficiently extended to consider N sets. The sweep scans the intervals across the sets to determine if any are in-context at the same time. In many data sets the number of N-way intersections is much smaller than the number of intervals, and a considerable amount of processing time can be saved by skipping regions of the input space that cannot possibly contain an intersection. Since the sweep algorithm must track each interval as it starts and ends, it is not able to skip over these regions. Furthermore, this type of serial processing has limited opportunities for parallel execution.

The "slice-then-sweep" strategy is a novel algorithm that can improve performance by both skipping regions that clearly lack N-way intersections, and exposing a significant amount of fine-grain parallelism. This algorithm efficiently *slices* the interval sets into independent regions so that only slices lacking empty regions are considered. Since the regions are inherently independent, they can also be easily processed in parallel. While the extent of the performance gain in the sequential case depends on the number of intervals that are excluded, the slice-then-sweep strategy performs as well as a linear sweep even in the worst case where none of the intervals may be excluded. Furthermore, the number of excluded intervals does not affect the level of parallelism that is achieved by slicing.

5.2.1 Sweep

The generalized sweep algorithm proposed by Bentley and Ottmann [29] is the basis for several pair-wise genomic interval intersection solutions [10, 14]. The two sorted interval sets are treated as stacks, and at each step the minimum interval between the sets (the current interval) is popped and added to an *ordering* data structure that maintains the set of intervals that are in-context. Intervals that end before the current interval starts are no longer in-context and are thus removed from the ordering. Intervals that intersect will be present in the ordering at the same time. The pair-wise sweep algorithm has been extended to solve the N-way intersection problem by adding an ordering for each of the interval sets and a priority queue to manage the removal of in-context intervals [10]. When all N orderings are not empty, the Cartesian product of the orderings gives a set of N-way intersections.

At each iteration, the N-way sweep (Algorithm 3) considers the interval with the next smallest start position across the ordered sets in $\mathbf{S}$. This process is similar to a merge operation and is managed by a priority queue Q where the minimum element in the queue has the highest priority. Each queue element $q \in Q$ is associated with an interval $x \in S_i$ and is assigned a priority equal to the interval start position ($q.p = x.start$) and a value equal to the index of the contributing set ($q.v = i$). Initially, the queue contains the first element in each set. As the sweep progresses, the next interval to be considered is determined by popping the highest priority element q_0 from Q. To replace this element with another that corresponds to the next element from the same set, an interval is popped from $S_{q_0.v}$, and a new element q_n is added to the queue where $q_n.p = x.start$ and $q_n.v = q_0.value$. Using a priority queue with $O(\log N)$ inserts and removes the time required to sweep N sets is proportional to $O(M \log N)$ were M is the total number of intervals among the sets.

Within the sweep, N ordering data structures $o_1 \dots o_N$ maintain the set of intervals that are in-context. Since the sets in S are free of contained intervals, each ordering data structure can be represented by the pair of indices $o_i = \langle o_i.start, o_i.end \rangle$ that specify the range of intervals in S_i that are in-context. For example, if $o_1 = \langle 3, 4 \rangle$ then $s_{1,3}$ and $s_{1,4}$ are in-context. If $o_i.start > o_i.end$, then none of the intervals in S_i are in-context. Each ordering is initialized to be empty ($o_i = \langle 0, -1 \rangle$ for $i = 1 \dots N$) and intervals are added by incrementing $o_i.end$ and removed by incrementing $o_i.start$.

When q_0 is popped from Q, any interval that is in-context and ends before $q_0.p$ (the start position of the interval associated with q_0) must be removed from context. Instead of scanning the orderings, which would

Algorithm 3: The N-way sweep algorithm.

```
Input: A set $\mathbf{S} = \{S_1, \ldots, S_N\}$ of ordered interval sets
Output: The set $N$-Way intersections by index
Function SWEEP begin
    $NW \leftarrow$ an empty set of $N$-way intersections
    $o \leftarrow$ set of $N$ pairs
    $Q \leftarrow$ priority queue for intervals to be swept
    $C \leftarrow$ priority queue for in-context intervals
    for $i \leftarrow 1$ to $N$ do
        $o_i.start \leftarrow 0$; $o_i.end \leftarrow -1$
        $x \leftarrow$ pop $(S_i)$
        $q_n.p \leftarrow x.start$; $q_n.v \leftarrow i$
        push $(Q, q_n)$
    $empty \leftarrow N$
    while $\neg$empty $(Q)$ do
        $q_0 \leftarrow$ pop $(Q)$
        if $\neg$empty $(S_{q_0.v})$ then
            $x \leftarrow$ pop $(S_{q_0.v})$
            $q_n.p \leftarrow x.start$; $q_n.v \leftarrow q_0.v$
            push $(Q, q_n)$
        while $q_0.p \geq$ top $(C)$ do
            $c \leftarrow$pop $(C)$
            $o_{c.v}.start \leftarrow o_{c.v}.start + 1$
            if $o_{c.v}.end < o_{c.v}.start$ then
                $empty \leftarrow empty + 1$
        if $o_{q_0.v}.end < o_{q_0.v}.start$ then
            $empty \leftarrow empty - 1$
        if $empty = 0$ then
            $nw \leftarrow \emptyset$
            for $i \leftarrow 1$ to $N$ do
                $s \leftarrow$sequence $(o_i.start, o_i.end)$
                $nw \leftarrow nw \times s$
            append $(NW, nw)$
        $o_{q_0.v}.end \leftarrow o_{q_0.v}.end + 1$
        $c_n.p \leftarrow S_{o_{q_0.v}.end}.end$
        $c_n.v \leftarrow q_0.v$
        push $(C, c_n)$
    return NW
```

require $O(N)$ time, another priority queue C is used to determine which orderings need to be updated in time $O(\log N)$ for each interval that is removed from context. Similar to Q, the minimum element in C has the highest priority, and the value of each element is the index of the contributing set. But unlike Q, the priority of the elements in C are interval end positions. To update the orderings, elements are popped from C until the top element has a priority greater than or equal to $q_0.p$. For every popped element c, the ordering $o_{c.v}$ is updated ($o_{c.v}$ += 1). An element associated the current interval is then added to C.

With the out-of-context intervals removed from the orderings, the interval corresponding to q_0 is placed in-context ($o_{q_0.v}$ += 1). If none of the orderings are empty, then an N-way intersection has been found. Each ordering represents the sequences of interval indices that are in-context. The Cartesian product of sequences defined by $o_1 \ldots o_N$ yields a set of N-way intersections. For example, if the orderings equal $\langle 1,1\rangle\langle 3,4\rangle\langle 2,2\rangle$, then the product gives a set of N-way intersections $\{(1,3,2),(1,4,2)\}$.

Within the scan each interval is added to and removed from the priority C exactly once, and the time required to scan a set of N interval sets with M total intervals is $O(M \log M)$. Since $N \leq M$, total time required to sweep a set the set is $O(M \log M + M \log N + W) = O(M \log M + W)$ where W is the number of N-way intersections.

5.2.2 Slice

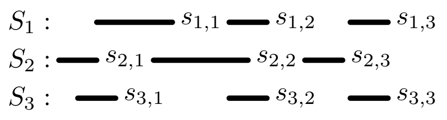
While the sweep algorithm is an efficient, general solution, it must consider *all* of the intervals in $\mathbf{S}$. As a result, when the number of intersections is much smaller than the number of intervals (as is common in the case of typical genomics datasets), a significant amount of time is spent sweeping regions of $\mathbf{S}$ that do not contain intersections. Here a new algorithm is proposed that attempts to minimize this type of over-processing by creating *slices* of $\mathbf{S}$. Any slice that cannot possibly contain an N-way intersection is discarded so that processing can focus on more promising regions. Slicing also creates independent subsets of $\mathbf{S}$ that can be processed in parallel.

Slices are composite sets that contain a subset of consecutive intervals from each set in $\mathbf{S}$. Since the intervals in each subset are consecutive, a slice $T = t_1 \ldots t_N$ can be represented by a set of start and end pairs $t_j = \langle t_j.start, t_j.end \rangle$, where $t_j.start$ denotes the index of the first interval of the slice from S_j and $t_j.end$ denotes the last. A subset is empty when $t_j.start > t_j.end$.

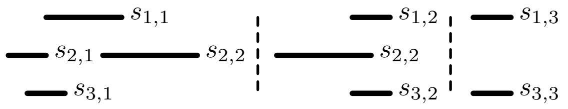
A slice is at *level* λ when the intervals in $t_{\lambda+1} \ldots t_N$ intersect all of the intervals in $t_1 \ldots t_\lambda$, and all of the intervals in $t_1 \ldots t_\lambda$ intersect each other. For example, the subsets in Figure 5.1b are level-one slices of the set in Figure 5.1a (which itself is considered a level-zero slice), and the subsets in Figure 5.1c are level-two slices. A slice at a particular level may contain an empty subset (such as the third slice in Figure 5.1b and the second slice in Figure 5.1c) if there are no intersecting intervals in the corresponding set. An interval may exist in multiple slices (such as interval $s_{2,2}$ in the first two level-one slices) if it intersects more than one interval in a higher level, and an interval may be excluded from any slice (such as interval $s_{2,3}$) if it doesn't intersect any higher-level intervals. A level-N slice with exactly one interval per subset is equivalent to an N-way intersection.

Basic slicing

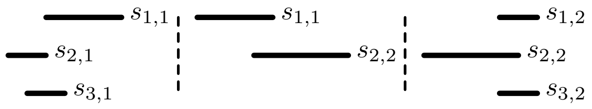
Considering that none of the sets in $\mathbf{S}$ have fully-contained intervals, a single level-one slice can be found with two binary searches per set for the start and end coordinates of a top-level interval $s_{1,i}$ (Algorithm 4). The first binary search (`binarySearchEnds`) finds the insert position of $s_{1,i}.start$ among the list of end coordinates in S_j, giving the index of the last interval in S_j to end before $s_{1,i}$ starts. The second binary



(a) The set $\mathbf{S}$



(b) The three level-one slices of $\mathbf{S}$



(c) The three level-two slices of $\mathbf{S}$

Figure 5.1: An example set of intervals sets $\mathbf{S}$, and level-one and -two slices of $\mathbf{S}$. Interval $s_{2,2}$ is in two level-one slices because it intersects both $s_{1,1}$ and $s_{1,2}$, and $s_{2,3}$ is not in any slice because it does not intersect any interval in S_1.

search (`binarySearchStarts`) finds the insert position of $s_{1,i}.end$ among the list of start coordinates in S_j, giving the index of the first interval in S_j to start after $s_{1,i}$ ends. The range between these two positions gives the intervals in S_j that intersect $s_{1,i}$, and the values to t_i. If that range is empty, then the start position will be greater than the end position.

Algorithm 4: The basic N-way slice operation.

```
Input: A set $\mathbf{S} = \{S_1, \ldots, S_N\}$ of ordered interval sets, and a base interval $s_{1,i} \in S_1$
Output: A level-one slice $C$
Function SLICE begin
    $C \leftarrow$ set of $N$ start and end pairs $\{c_1, \ldots, c_N\}$
    $c_1.start \leftarrow i$; $c_1.end \leftarrow i$
    for $j \leftarrow 2$ to $N$ do
        $left \leftarrow$ binarySearchEnds ($s_{1,i}.start, S_j$)
        $right \leftarrow$ binarySearchStarts ($s_{1,i}.end, S_j$)
        $c_j.start \leftarrow left + 1$; $c_j.end \leftarrow right - 1$
    return $C$
```

The time required to compute all level-one slices for each interval in S_1 is $O(|S_1|\log(NM))$, where $|S_i|$ is the number of intervals in S_i, N is the number of interval sets in $\mathbf{S}$, and M is the number of intervals in $\mathbf{S}$. The creation of each subset of each slice takes time $O(\log(|S_i|))$, and the time to create a full slice is $O(\sum \log(|S_i|)) = O(\log(\prod |S_i|)) = O(\log(NM))$ (given that $\prod_{i=1}^{N} |S_i| \leq NM$). There are $|S_1|$ level-one slices, making total time $O(|S_1|\log(NM))$.

Pivot slicing

While the basic slicing process is reasonably efficient, when level-one slices are created in a particular order information about the distribution of intervals can be utilized by future slicing operations to both reduce the search space and possibly prevent some slices from being created. For example, consider the set in Figure 5.2. In the basic process, three level-one slices are created ($\{\{s_{1,1}\}, \{\}, \{s_{3,1}\}\}$, $\{\{s_{1,2}\}, \{\}, \{s_{3,2}\}\}$, and $\{\{s_{1,3}\}, \{s_{2,1}, s_{2,2}\}, \{s_{3,2}\}\}$). Two of the these slices contain an empty subset (the two based on intervals

$s_{1,1}$ and $s_{1,2}$), and thus they cannot possibly be part of an N-way intersection. If the process instead started by creating the slice centered on $s_{1,2}$, then it could have been inferred that the slice on $s_{1,1}$ would contain an empty subset (since $s_{1,1} \leq s_{1,2}$ and $s_{2,i} > s_{1,2}$ for $i = 1, \ldots, |S_2|$), and that it did not need to be created since it could not possibly lead to an N-way intersection.

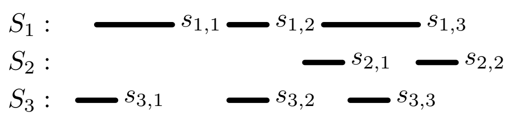


Figure 5.2: If the sliced based on $s_{1,2}$ is created first, then it can be inferred that $s_{1,1}$ is not part of an N-way intersection and a slice based on it does not need to be created.

The "emptiness" of future slices can be inferred by systematically creating slices based on a *pivot interval* $s_{1,p} \in S_1$ (Algorithm 5). The algorithm is similar to the previous slicing method, requires less work, and in many instances results in fewer slices. With respect to the pivot (e.g., the middle interval), the sets in $\mathbf{S}$ are partitioned into *left*, *center*, and *right* slices. The left slice includes intervals that are less than or equal to $s_{1,p}$, the center slice includes intervals that intersect $s_{1,p}$, and the right slice includes intervals that are greater than or equal to $s_{1,p}$. Any slice containing an empty subset is discarded. Non-empty left and right slices are recursively re-sliced in a breadth-first search style algorithm (Algorithm 6), and the center slices make up the set of level-one slices. To find the N-way intersections, level-one slices can be similarly re-sliced into left, center, and right slices, and the center slices are then level-two slices. The center slices of level-two slices are level-three slices and are re-sliced into level-four slices, and so on until level-N. After level-N, all non-empty center slices give the N-way intersections.

Algorithm 5: The left, center, and right N-way slice operation.

```
Input: A set $\mathbf{S} = \{S_1, \ldots, S_N\}$ of ordered interval sets, a slice $T = \{t_1, \ldots, t_N\}$, and a pivot interval $s_{1,p}$ where
       $t_1.start \leq p \leq t_1.end$
Output: The list of left, center, and right slices $[L, C, R]$
Function LCRslice begin
    $L \leftarrow$ set of $N$ start and end pairs $\{l_1, \ldots, l_N\}$
    $C \leftarrow$ set of $N$ start and end pairs $\{c_1, \ldots, c_N\}$
    $R \leftarrow$ set of $N$ start and end pairs $\{r_1, \ldots, r_N\}$
    $l_1.start \leftarrow 1$; $l_1.end \leftarrow p - 1$
    $c_1.start \leftarrow p$; $c_1.end \leftarrow p$
    $r_1.start \leftarrow p + 1$; $r_1.end \leftarrow |S_1|$
    for $j \leftarrow 2$ to $N$ do
        $left \leftarrow$binarySearchEnds $(s_{i,p}.start, S_j, t_j.start, t_j.end)$
        $right \leftarrow$binarySearchStarts $(s_{i,p}.end, S_j, t_j.start, t_j.end)$
        $l_j.start \leftarrow t_j.start$; $l_j.end \leftarrow right - 1$
        $c_j.start \leftarrow left + 1$; $c_j.end \leftarrow right - 1$
        $r_j.start \leftarrow left + 1$; $r_j.end \leftarrow t_j.end$
    return $[L, C, R]$
```

To allow for re-slicing, LCRslice is defined in terms of a slice T and a pivot in the first subset of T. The binary searches are also modified to operate only within the bounds of the current slice. This reduces

Algorithm 6: The pivot slice algorithm.

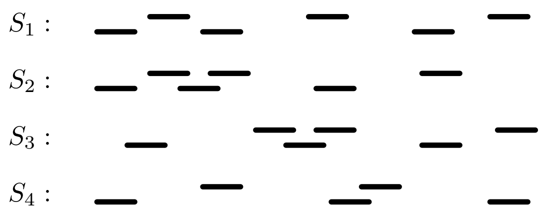
```
Input: A set $\mathbf{S} = \{S_1, \ldots, S_N\}$ of ordered interval sets
Output: The list of level-one slices $L$
Function PivotSlice begin
    $Q \leftarrow$ a queue of slices
    $L \leftarrow$ an empty set of level-one slices
    $T \leftarrow$ set of $N$ start and end pairs
    for $i \leftarrow 1$ to $N$ do
        $t_i.start \leftarrow 1$; $t_i.end \leftarrow |S_i|$
    push $(Q, T)$
    while $q \leftarrow$pop $(Q)$ do
        $p \leftarrow (q_1.start + q_1.end + 1)/2$
        $[L, C, R] =$LCRslice$(\mathbf{S}, s_{1,p})$
        if $L$ is not empty then
            push $(Q, L)$
        if $R$ is not empty then
            push $(Q, R)$
        if $C$ is not empty then
            append $(L, C)$
    return $L$
```

the amount of work required to find all slices since only the first round of slicing considers all intervals in $\mathbf{S}$ ($t_i.start = 1$ and $t_i.end = |S_i|$ for $i = 1 \ldots N$), and subsequent rounds consider increasingly smaller subsets of $\mathbf{S}$. At each iteration slices are roughly divided in half by setting the pivot interval to the medoid interval in t_1.

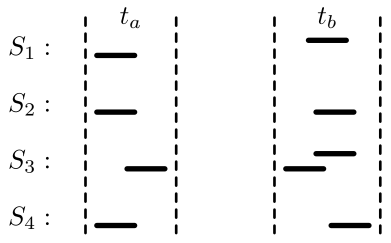
Slice and sweep

N-way intersections can be found by re-slicing each level-one slice into level-two slices, then level-three slices, and so on. Following this procedure, level-N slices will contain exactly one interval per subset, and all the intervals will intersect each other. However, as with the sweep operation, re-slicing level-one slices may result in over-processing.

The slice operation is best suited for cases where the proportion of intervals involved in an N-way intersection is much smaller than the total number of intervals. While this property exists in many real-world data sets, it may not continue to exist in the level-one slices of the data. Consider the full set and the level-one slices in Figure 5.3. The non-empty, level-one slices contain less than half the number of intervals that are in the full set. If these two slices were then re-sliced, the resulting level-two slices would contain the exact same number of intervals as the level-one slices and only one additional interval would have been added to the possible N-way intersections. To obtain the final result, such wasted processing would continue through the remaining sets with a runtime total runtime of $O(WN \log(NM))$, where W is the total number of N-way intersections.



(a) The set **S**.



(b) The two non-empty, level-one slices of **S**.

Figure 5.3: The non-empty level-one slices comprise less than half of the full set.

Considering that over-processing occurs in the sweep when the proportion of intervals in N-way intersections to the total number of intervals is low, and in the slice when the proportion is high, it follows that the most efficient algorithm is a hybrid that intelligently switches from slicing to sweeping. Conceptually, slicing should stop before the amount of work required to slice is greater than the amount of work required to sweep the intervals that would have been excluded by the slice. Since determining exactly when this transition will occur requires *a priori* knowledge of the final result (i.e., the total number of the N-way intersections in the slice), a heuristic must be used.

While the number of intervals that would be removed by performing an additional level of slicing is unknown, it is obviously bound by the total number of intervals in the slice. To account for the varying number of sets, the decision to slice or sweep can be based on the ratio of the number of intervals in the slice to the number of sets. As the ratio approaches one, the potential for the slicing algorithm to remove intervals decreases. Outside of pathological cases (very long intervals or very dense collections of intervals), the ratio drops to near one after the first level of slicing. Based on this observation, the slice and sweep algorithms presented here switch to sweeping after one level of slicing. If the ratio for a slice is exactly one (there is only one interval per subset) then a sweep in not needed and the intervals in each slice are simply tested to see if they form an N-way intersection. The runtime of this algorithm is equal to the time required to create the level-one slices plus the time required to sweep those slices, which is $O(|S_1|\log(NM) + M'\log(M') + W)$, where M' is the number of intervals that were not excluded by the slice.

Parallel slice and sweep

Beyond reducing the over-processing in the sweep algorithm, slicing also creates inherently independent sections of **S** that can be processed in parallel. While pivot slicing reduces the amount of work required to create a set of level-one slices, the basic slicing algorithm is a better candidate for parallelization since each slice is completely independent (pivot slicing creates new slices at each step, creating a dependency between slices) and there are no shared resources (pivot slicing has a queue that must be locked on push and pop). In the parallel slice and sweep algorithm, each thread creates one level-one slice. Then, any non-empty level-one slice is swept. Each thread maintains a private list of N-way intersections that are merged in the final step of the algorithm.

5.3 Results

To assess the performance of these algorithms, the sweep (sweep), pivot-slice (pslice), pivot slice then sweep (pslice-sweep), and parallel slice then sweep that is based on the basic slice operation (Pbslice-sweep) algorithms were implemented as stand-alone C utilities using the Pthreads library for parallel thread manipulation. As pointed out in the previous section, the relative performance of each algorithm depends on the distribution of the data. As the proportion of intervals involved in an N-way intersection increases, the opportunities for the split algorithm to reduce the amount of data that must be considered diminishes. To identify where slicing no longer provides improvement over sweeping, the performance of the algorithms were tested against simulated data sets with a varying numbers of sets and varying proportion of N-way intersections. Each algorithm was also tested using previously published data from a large-scale DNase I hypersensitive site study [30], and CTCF transcription factor sites from the ENCODE project [31].

All tests were performed on an Intel Xeon(R) CPU with eight 2.6 GHz cores (16 threads), and 20 MB of cache. The system was running Red Hat Linux version version 2.6.32-358.6.1.el6.x86_64, with gcc version 4.4.7-3. Unless otherwise noted, the results do not include disk read or write time. The software tested here is freely available at `https://github.com/ryanlayer/nway`.

5.3.1 Performance with Simulated Interval Sets

To test the extent to which the slice algorithms can improve performance over the sweep algorithm, each was tested against uniformly distributed random data sets with varying numbers of sets and varying proportions of N-way intersections. Each set within the test contained 10,000 intervals, each interval was 100 base pairs long, and all intervals were placed within the genome. The number of sets ranged from 10 to 200, and the proportion of N-way intersections ranged from 1% to 100%.

In each test, the sites of the intersections were randomly generated. Then for each set and for each site, an interval was generated to contain the site at a random point within the interval. The remaining non-intersecting intervals were then added at random locations within the range. Each test was executed three times and the mean runtime in microseconds is given in Figure 5.4.

As expected, the benefit of slicing as highest when the proportion of intervals in an N-way intersection was low. At 1%, all of the slice-based algorithms outperformed the sweep, including the slice-only algorithm which is known to over process. The pivot slice-sweep algorithm was up to three times faster than sweep, and the sequential slice-then-sweep was nearly seven times faster. The parallel slice-then-sweep had a dramatic improvement with a 25 to 30 fold speedup over the sequential sweep.

When the proportion of intersections increases, the opportunities for slicing to reduce the amount of work decreases. As a result, when the density of intersections increases, the improvement of the slice algorithms decreases. However, the sequential slice-then-sweep algorithm performed as well as or better than the sweep in every case, and the parallel slice-then-sweep was up to 7.5 times faster when 50% of the intervals were in N-way intersections, and up to 5 times faster when all intervals were in an N-way intersect.

The over processing in the slice-only algorithm was clearly demonstrated in many of the larger tests where the runtime was greater than the plot area (denoted by a "*"). Similarly, the extra work that is required to find level-one slices using the basic slicing method resulted in poor performance when using a single thread.

5.3.2 Performance with Published Genomic Interval Sets

In an attempt to understand the function of specific regions of the human genome, large-scale studies are continuously publishing interval sets that describe the locations of particular genomic features and cellular functions across many different cell types. The experiments used to find these sites can have a significant amount of noise and each experiment can result in hundreds of thousands of sites. By comparing results among many experiments, researchers can narrow the search space to the most promising regions and develop new hypotheses and future experiments that focus on these regions.

Maurano et al. [30] published sites of DNase I hypersensitivity observed among more than 200 fetal tissue samples (e.g., lung, heart, kidney). These sites mark regions of the genome that are accessible to cellular processes such as DNA binding proteins and are thought to be active regions of the genome. The interval sets have on average 2e5 intervals per set, intervals have a mean length of 500, and there are 15,340 N-way intersections across the 200 datasets (7.5% of intervals in an intersection).

The ENCODE project has published thousands of interval sets [31], including CTCF binding sites from over 100 different cell lines (e.g., the myelogenous leukemia line K562, and the cervical cancer line Hela). CTCF is a transcription factor that has been shown to have a role in repressing cellular activity. The interval sets have on average 1.8e5 intervals per set, intervals have a mean length of 215, and there are 17,071 N-way intersections across the 100 datasets (9.5% of intervals in an intersection).

Considering the data set sizes and the proportion of intersecting intervals in these data sets, the runtime improvements of the slice algorithms over the sweep algorithm were consistent with the results from the simulated data. In the DNase I hypersensitive sites data set, the slice-then-sweep algorithm was nearly twice as fast as sweep, and the parallel versions of slice-then-sweep were 5x, 10x and 19x faster for four, eight, and 16 threads, respectively (Figure 5.5). While the CTCF data did not have as many data sets (100 vs. 200 for DNase I and the simulations) the speedup trends was still consistent, with the slice-then-sweep 1.5x faster and the parallel slice-then-sweep being 6x, 10x, and 19x faster with four, eight, and 16 threads, respectively (Figure 5.6). Both data sets also demonstrated the issues with wasted processing in the slice algorithm.

5.4 Conclusion

In order to improve upon the efficiency of linear sweep approaches to interval intersection, the slice-then-sweep algorithm employs a hybrid strategy that is designed to find intersections across all N data sets. Given its inherent potential for parallelism by partitioning interval sets into independent subsets, slice-then-sweep provides an approach that scales to the size and complexity of modern genomics datasets. When considering real data, slicing cuts the sweep runtime in half when using a single thread, and is up to 19x faster when using multiple threads. By vastly expanding the number of sets that can be considered, the slice-then-sweep algorithm, along with BITS from the previous chapter, provide researchers with the ability to generate new hypotheses that are based on the relationships between genomic features.

Efficient interval intersection is both a means to compare genomic features, and a platform from which to build higher-level genomic analysis algorithms. High-throughput genomic sequencing produces billions of reads from a single sample, and when aligned to a reference genome, billions of genomic intervals. Each of these intervals carries information about the structure of the samples genome. In the following chapter, the LUMPY algorithm uses an interval intersection framework to detect structural variations in the sample's genome from high-through sequencing data. Considering these variations have been linked to a broad range of diseases, from cancer to autism, this is one of the most important problems in genomics.

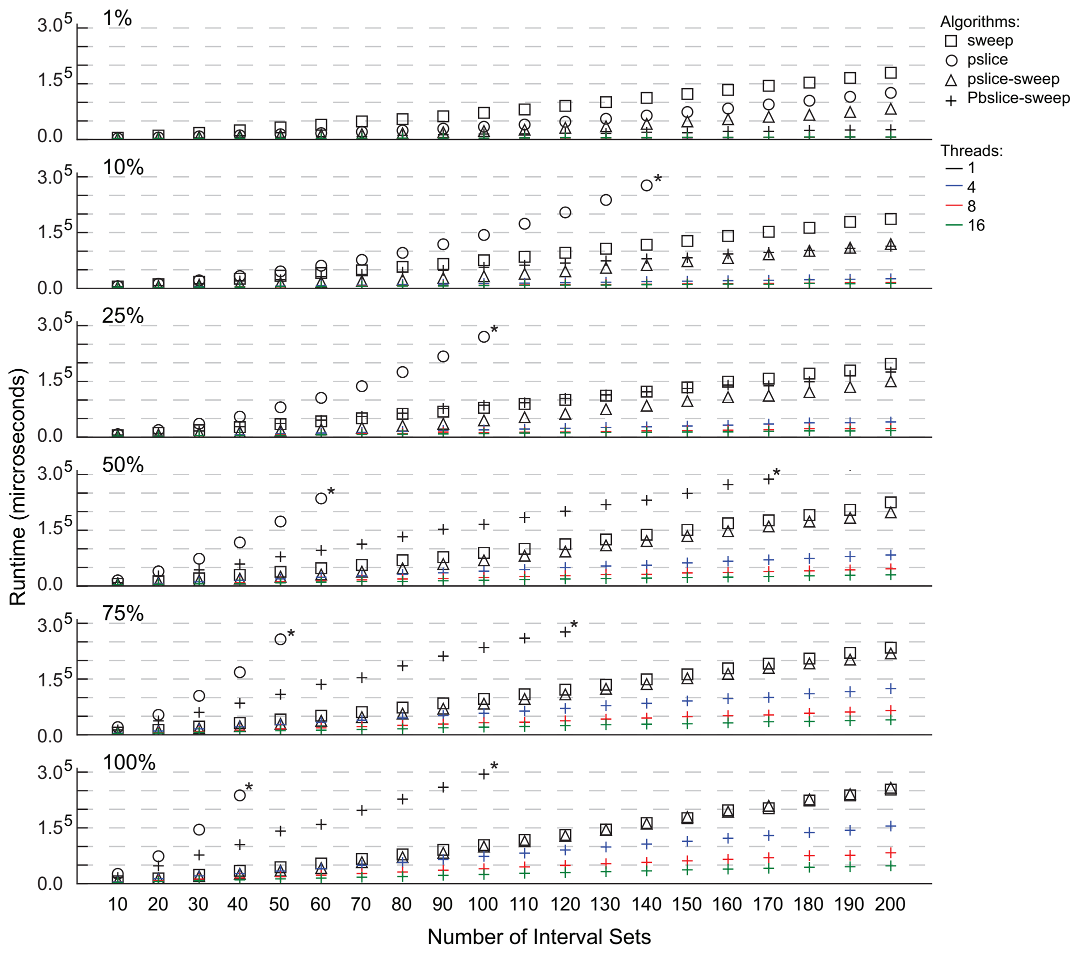


Figure 5.4: The runtime performance of N-way intersection problems on simulated data. The proportion of intervals in an intersection increases from 1% to 100% from top to bottom. Data series marked by an "*" have runtimes that exceed the limits of the graph in subsequent cases.

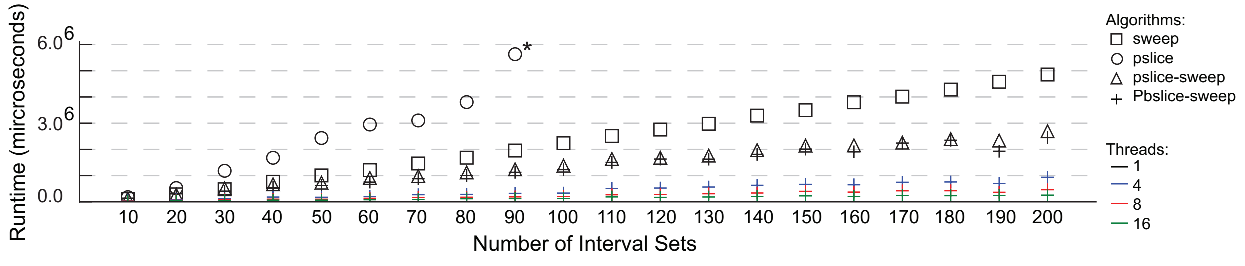


Figure 5.5: The runtime performance of N-way intersection problems on DNase I hypersensitive sites. Data series marked by an "*" had runtimes that exceed the limits of the graph in subsequent cases.

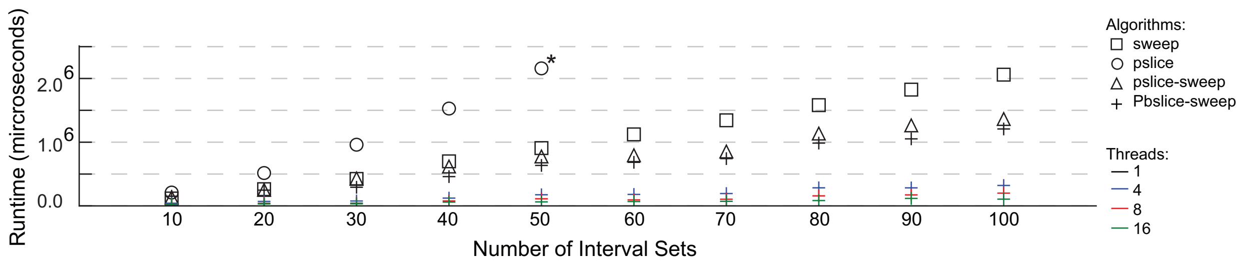


Figure 5.6: The runtime performance of N-way intersection problems on CTCF binding sites. Data series marked by an "*" had runtimes that exceed the limits of the graph in subsequent cases.

Chapter 6

LUMPY: A Probabilistic Framework for Structural Variant Discovery

6.1 Introduction

Modern high-throughput paired-end DNA sequencing technologies are able to produce billions of reads from a single sample. Each read contains information about the composition of the source's genome. Structural variations (SVs) that exist in the sample can be detected by analyzing the full set of reads. This type of analysis is particularly important since many diseases have been linked to genetic alterations. The issue is that the majority of the sequencing platforms currently in use produce reads that are only about 100 bases long, which is a tiny faction of the full genome (the human genome is over 3 billion bases long). This, along with a number of stochastic errors associated with the sequencing process makes it difficult to distinguish a true SV signal from the noise. Therefore, the comprehensive discovery of human genome SVs from sequencing data requires the integration of all available signals (see Section 2.3). However, owing to inherent technical challenges, existing SV discovery algorithms either use a single signal in isolation, or at best use two signals sequentially. Such approaches suffer from limited sensitivity when a variant is captured by few reads, as often occurs in low coverage datasets or at somatic mutations with low intra-sample allele frequencies. In particular, sensitive detection of low frequency somatic SVs in heterogeneous tumor samples is an unsolved problem with clinical implications.

This thesis presents LUMPY, a novel and extremely flexible probabilistic SV discovery framework capable of integrating any number of SV detection signals, including those generated from read alignments or prior evidence, jointly across multiple samples. LUMPY enables straightforward signal integration by mapping

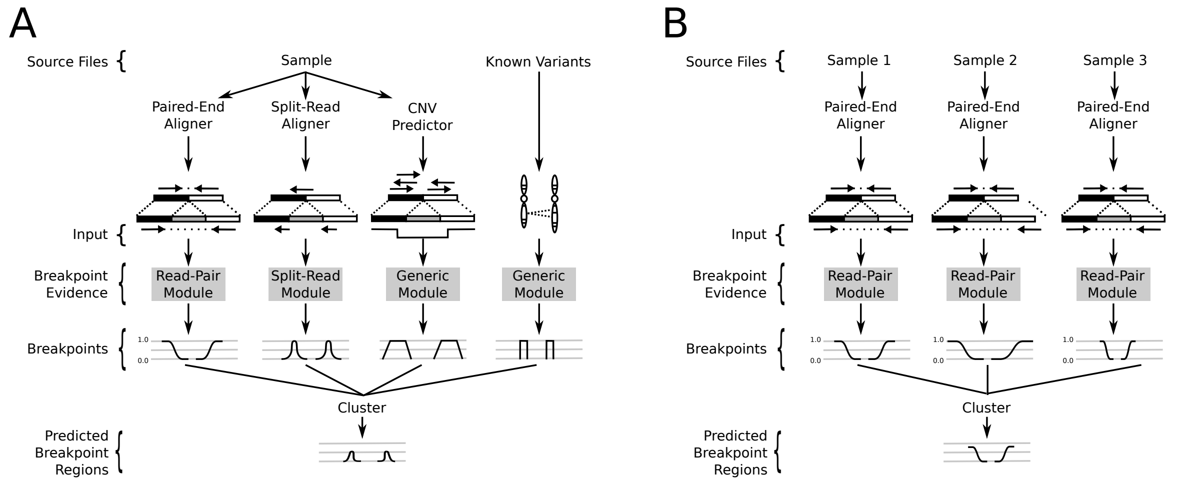


Figure 6.1: LUMPY workflows. (a) A LUMPY workflow using three different signals (read-pair, split-read and read-depth) from one sample, as well as prior knowledge regarding known variant sites. (b) A LUMPY workflow using a single signal type (in this case, read-pair) from three different samples. Note that multi-signal and multi-sample workflows are not mutually exclusive.

each SV signal to a common abstract representation in the form of breakpoint probability distributions and then performs SV prediction operations at this higher level. This approach allows for simple and natural signal integration, inherently produces a probabilistic measure of breakpoint position, and can be easily extended to new signals as sequencing technologies and SV detection strategies evolve. Results from both simulated and real data demonstrate that integration of read-pair and split-read signals yields improved sensitivity over extant methods, especially when the SV signal is reduced owing to either low coverage or low variant frequency in heterogeneous samples such as cancer.

6.2 Methods

6.2.1 Overview

The LUMPY framework is based upon a general probabilistic representation of an SV breakpoint that allows any number of alignment signals to be integrated into a single discovery process. A breakpoint is defined as a pair of bases that are adjacent in an experimentally sequenced “sample” genome but not in the reference genome. To account for the varying level of genomic resolution inherent to different types of alignment evidence, a breakpoint is represented with a pair of probability distributions spanning the predicted breakpoint regions (Figure 6.1). The probability distributions reflect the relative uncertainty that a given position in the reference genome represents one end of the breakpoint.

This framework provides distinct modules that map signals from each alignment evidence type to the common probability interval pair. For example, paired-end sequence alignments are projected to a pair of intervals upstream or downstream (depending on orientation) of the mapped reads (Figure 6.1a). The size of the intervals and the probability at each position is based on the empirical size distribution of the sample's DNA fragment library. The distinct advantage of this approach is that any type of evidence can be considered, as long there exists a direct mapping from the SV signal to a breakpoint probability distribution. Here, three modules are provided for converting SV signals to probability distributions: read-pair, split-read, and generic. The framework is also readily extensible to new signals that may potentially result from new DNA sequencing technologies or alternative SV detection approaches. The read-pair module maps the output of a paired-end sequence alignment algorithm such as NOVOALIGN [32] or BWA [5], the split-read module maps the output of a split-read sequence alignment algorithm (e.g., YAHA [33], BWA-SW [34], or BWA-MEM [35], and the generic module allows users to include SV signal types that do not have a specific module implemented (e.g., *a priori* knowledge such as known SV, and/or output from copy number variation discovery tools).

Once the evidence from the different alignment signals is mapped to breakpoint intervals, overlapping intervals are clustered and the probabilities are integrated. Any clustered breakpoint region that contains sufficient evidence (based on user-defined arguments) is returned as a predicted SV. The resolution of the predicted breakpoint regions is improved by trimming the positions with probabilities in the lower percentile of the distribution (e.g., the lowest 0.1 percent).

It is well established that variant calling is improved by integrating data from multiple samples [36, 37, 38, 39]. The LUMPY framework naturally handles multiple samples by tracking the sample origin of each probability distribution during clustering (Figure 6.1b). As an example of a typical analysis, LUMPY can identify SVs in a whole-genome, 50X coverage paired-end Illumina dataset from the NA12878 CEPH individual in 12.2 hours using 8 Gb of memory using a single processor. Given that these performance characteristics are comparable to existing approaches for SNP and INDEL detection, and that there is an approximately linear relationship between data volume, time and memory usage, it is anticipated that simultaneous analysis of tens and eventually hundreds of human genomes will be possible with LUMPY using commodity hardware.

LUMPY in an open source C++ software package (available at `https://github.com/arq5x/lumpy-sv`) that is capable of detecting structural variation from multiple alignment signals in BAM [5] files from one or more samples.

6.2.2 Breakpoint

A breakpoint is a pair of genomic coordinates that are adjacent in a sample genome but not in a reference genome. Breakpoints can be detected, and their locations predicted by various evidence classes such as paired-end sequence alignments or split-read mappings. To support the inclusion of different evidence classes into a single analysis, LUMPY defines a high-level breakpoint type as a collection of the evidence that corroborates the location and variety (e.g., deletion, tandem duplication, etc.) of a particular breakpoint. Since many evidence classes provide a range of possible locations, a breakpoint's location is represented with a pair of breakpoint intervals where each interval has a start position, an end position, and a probability vector that represents the relative probability that a given position in the interval is one end of the breakpoint. More formally, a breakpoint is a tuple $b = \langle E, l, r, v\rangle$, where $b.E$ is the set of evidence that corroborates the location and variety of a particular breakpoint; $b.l$ and $b.r$ are left and right breakpoint intervals each with values $b.l.s$ and $b.l.e$ that are the start and end genomic coordinates and $b.l.p$ is a probability vector where $|b.l.p| = b.l.e - b.l.s$ and $b.l.p[i]$ is the relative probability that position $b.l.s + i$ is one end of the breakpoint (similar for $b.r$); and $b.v$ is the breakpoint variety. Within the context of this method, breakpoint variety determinations are based on the orientation of the evidence. It is important to note that while a breakpoint may be labeled as a deletion when it contains evidence from a paired-end sequence alignment with a +/- orientation, the breakpoint may in fact be the result of some other event or series of complex events.

If there exist two breakpoints b and c in the set of all breakpoints B where b and c intersect ($b.r$ intersects $c.r$, $b.l$ intersects $c.r$, and $b.v = c.v$), then b and c are merged into interval m, b and c are removed from B, and m is placed into B. The evidence set $m.E$ is the union of the evidence sets $b.E$ and $c.E$.

A straight-forward method to define breakpoint intervals $m.l$ and $m.r$ would be to let $m.l.s = \max(b.l.s, c.l.s)$ and $m.l.e = \min(b.l.e, c.l.e)$, similar for $m.r$. However, if a spurious alignment is merged into a set of genuine breakpoints, the resulting breakpoint interval can be pulled away from the actual breakpoint. The impact of an outlier can be minimized or eliminated once the full set of corroborating alignments is collected for a given breakpoint, but collecting the full set is complicated by the fact that alignments are considered in order and outliers typically occur first. To account for this a liberal merge process is used where $m.l.s$ is the mean start position for the left intervals in $m.E$, and $m.l.e$ is the mean end position for the left intervals in $m.E$, similar for $m.r$.

Once all the evidence has been considered, an SV call s (also a breakpoint) is made for each breakpoint $b \in B$ that meets a user-defined minimum evidence threshold (e.g., four pieces of evidence). The boundaries of the breakpoint intervals $s.l$ and $s.r$ are the trimmed mixture distributions of the left and right intervals in $b.E$. Let $s.l.s = \max(\{e.l.s|e \in b.E\})$, $s.l.e = \min(\{e.l.e|e \in b.E\})$, and $s.l.p[i] = \sum_{e \in b.E} e.l.p[i - o]$ where o is the offset value $e.l.s - s.l.s$ (similar for $s.r$). The value at $s.l.p[i]$ (or $s.r.p[i]$) represents the level of agreement among the evidence in $b.E$ that position i is one end of the breakpoint. The intervals $s.l$ and $s.r$ are then trimmed to include only those positions that are in the top percentile (e.g., top 99.9 percent of values). An outlier in $b.E$ will extend the interval $s.l$, but the extended region will have little support from other elements in $b.E$ and values of $s.p$ in that region will be relatively small and are likely to be removed in the trimming process.

6.2.3 Breakpoint Evidence

LUMPY uses an abstract breakpoint evidence type to combine distinct SV alignment signals such as read-pair and split-read alignments with the general breakpoint type defined above. This abstract type defines an interface that allows for the inclusion of any data that can provide the following functions: `is_bp` determines if a particular instance of the data contains evidence of a breakpoint, `get_v` determines the breakpoint variety (e.g., deletion, tandem duplication, inversion, etc.), and `get_bpi` maps the data to a pair of breakpoint intervals.

To demonstrate the applicability of this abstraction, three breakpoint evidence instances are given: paired-end sequencing alignments, split-read alignments, and a general breakpoint interface. Read-pair and split-read are among the most frequently used evidence types for breakpoint detection, and the general interface provides a mechanism to include any other sources of information such as known variant positions or output from other analysis pipelines (e.g., read-depth calls). As technologies evolve and the understanding of structural variation improves, other instances can be easily added.

Paired-End Alignments

Paired-end sequencing involves fragmenting genomic DNA into roughly uniformly-sized fragments and sequencing both ends of each fragment to produce paired-end reads $\langle x, y\rangle$, which will be referred to as "read-pairs". Each read is aligned to a reference genome $R(x) = \langle c, o, s, e\rangle$, where $R(x).c$ is the chromosome that x aligned to in the reference genome, $R(x).o = +|-$ indicates the alignment orientation, and $R(x).s$ and $R(x).e$ delineate the start and end positions of the matching sequence within the chromosome. It is assumed that both x and y align uniquely to the reference and that $R(x).s < R(x).e < R(y).s < R(y).e$. While in practice it is not possible to know the position of read x in the sample genome (in the absence of whole-genome assembly), it is useful to refer to $S(x) = \langle o, s, e\rangle$ as the alignment of x with respect to the originating sample's genome.

Assuming that genome sequencing was performed with the Illumina platform, read-pairs are expected to align to the reference genome with a $R(x).o = +$, $R(y).o = -$ orientation, and at distance $R(y).e - R(x).s$ roughly equivalent to the fragment size from the sample preparation step. Any read-pair that aligns with an unexpected configuration can be evidence of a breakpoint. These unexpected configurations include matching orientation $R(x).o = R(y).o$, alignments with switched orientation $R(x).o = -$, $R(y).o = +$, and an apparent fragment length ($R(y).e - R(x).s$) that is either shorter or longer than expected. The expected fragment length is estimated to be the sample mean fragment length L, and the fragment length standard deviation to be the sample standard deviation s from the set of properly mapped read-pairs (as defined by the SAM specification) in the sample data set. Considering the variability in the sequencing process, the expected fragment length is extended to include sizes $L + v_l s$, where v_l is a tuning parameter that reflects spread in the data.

When x and y align to the same chromosome ($R(x).c = R(y).c$), the breakpoint variety can be inferred from the orientation of $R(x)$ and $R(y)$. If the orientations match, then the breakpoint is labeled as an inversion, and if $R(x).o = -$ and $R(y).o = +$ then the breakpoint is labeled as a tandem duplication. Any breakpoint with the orientation $R(x).o = +$ and $R(y).o = -$ is labeled as a deletion. When x and y align to different chromosomes ($R(x).c \neq R(y).c$), the variety is labeled inter-chromosomal. At present, LUMPY does not explicitly support identification of insertions that are spanned by paired-end reads, however, if desired these can be identified in a post-processing step through assessment of "deletion" calls.

To map $\langle x, y \rangle$ to breakpoint intervals l and r, the ranges of possible breakpoint locations must be determined and probabilities assigned to each position in those ranges. By convention, x maps to l and y to r, and for the sake of brevity the focus will be on x and l since the same process applies to y and r. Assuming that a single breakpoint exists between x and y, then the orientation of x determines if l will be upstream or downstream of x. If the $R(x).s = +$, then the breakpoint interval begins after $R(x).e$ (downstream), otherwise the interval ends before $R(x).s$ (upstream).

The length of each breakpoint interval is proportional to the expected fragment length L and standard deviation s. Since it is assumed that only one breakpoint exists between x and y, and that it is unlikely that the distance between the ends of a pair in the sample genome ($S(y).e - S(x).s$) is greater than L, then it is also unlikely that one end of the breakpoint is at a position greater than $R(x).s + L$, assuming that $R(x).o = +$. If $R(x).o = -$, then it is unlikely that a breakpoint is at a position less than $R(x).e - L$. To account for variability in the fragmentation process, the breakpoint is extended to $R(x).e + (L + v_f s)$ when $R(x).o = +$, and $R(x).s - (L + v_f s)$ when $R(x).o = -$, where v_f is a tuning parameter that, like v_l, reflects the spread in the data.

The probability that a particular position i in the breakpoint interval l is part of the actual breakpoint can be estimated by the probability that x and y span that position in the sample. For x and y to span i, the fragment that produced $\langle x, y \rangle$ must be longer than the distance from the start of x to i, otherwise y would occur before i and x and y would not span i (contradiction). The resulting probability is $P(S(y).e - S(x).s > i - R(x).s)$ if $R(x).o = +$, and $P(S(y).e - S(x).s > R(x).e - i)$ if $R(x).o = -$. While one cannot directly measure the sample fragment length ($S(y).e - S(x).s$), it is possible to estimate its distribution by constructing a frequency-based cumulative distribution D of fragment lengths from the same sample that was used to find L and s, where $D(j)$ gives the proportion of the sample with fragment length greater than j.

Split-Read Alignments

A split-read alignment occurs when a single DNA fragment X does not contiguously align to the reference genome. Instead, X contains a set of two or more substrings $x_i \dots x_j$ ($X = x_1 x_2 \dots x_n$), where each substring aligns to the reference $R(x_i) = \langle c, o, s, e \rangle$, and adjacent substrings align to non-adjacent locations in the reference genome $R(x_i).e \neq R(x_i+1).s+1$ or $R(x_i).c \neq R(xi+1).c$ for $1 \leq i \leq n1$. A single split-read alignment maps to a set of adjacent split-read sequence pairs $(\langle x_1, x_2 \rangle, \langle x_2, x_3 \rangle, \dots, \langle x_{n-1}, x_n \rangle)$, and each pair $\langle x_i, x_{i+1} \rangle$ is considered individually.

By definition, a split-read mapping is evidence of a breakpoint and therefore the function `is_bp` trivially returns true.

Both orientation and mapping location must be considered to infer the breakpoint variety for $\langle x_i, x_{i+1} \rangle$. When the orientations match $R(x_i).o = R(x_i+1).o$, the event is marked as either a deletion or a tandem duplication. Assuming that $R(x_i).o = R(x_i+1).o = +$, $R(x_i).s < R(x_i+1).s$ indicates a gap caused by a deletion and $R(x_i).s > R(x_i+1).s$ indicates a tandem duplication. These observations are flipped when orientations $R(x_i).o = R(x_i+1).o = -$. When the orientations do not match $R(x_i).o \neq R(x_i+1).o$, the event is marked as an inversion and the mapping locations do not need to be considered. When x and y align to different chromosomes, the variety is marked as inter-chromosomal. LUMPY does not currently attempt to identify insertions that are completely contained within a long read, but this will be supported in future versions. This capability requires long-read aligners to report the number and order of alignments within a read (which is not formally supported in the current SAM format specifications).

The possibility of errors in the sequencing and alignment processes creates some ambiguity in the exact location of the breakpoint associated with a split-read alignment. To account for this, each alignment pair $\langle x_i, x_i+1 \rangle$ maps to two breakpoint intervals l and r centered at the split. The probability vectors $l.p$ and $r.p$ are highest at the midpoint and decrease exponentially toward their edges. The size of this interval is a configurable parameter v_s and is based on the quality of the sample under consideration and the specificity of the alignment algorithm used to map the sequences to the reference genome.

Depending on the breakpoint variety, the intervals l and r are centered on either the start or the end of $R(x_i)$ and $R(x_i+1)$. When the breakpoint is a deletion l is centered at $R(x_i).e$ and r at $R(x_i+1).s$, and when the breakpoint is a tandem duplication l is centered at $R(x_i).s$ and r at $R(x_i+1).e$. If the breakpoint is an inversion, l and r are both centered either at the start positions or end positions of $R(x_i)$ and $R(x_i+1)$, respectively. Assuming that $R(x_i).s < R(x_i + 1).s$, if $R(x_i).o = +$ then l and r are centered at $R(x_i).e$ and $R(x_i + 1).e$, otherwise they are centered at $R(x_i).s$ and $R(x_i + 1).s$. If $R(x_i).s > R(x_i + 1).s$, then the conditions are swapped.

Generic Evidence

The generic evidence subclass provides a mechanism to directly encode breakpoint intervals using the BEDPE format [10]. BEDPE is an extension of the popular BED format that provides a means to specify a pair of genomic coordinates; in this case the pair represents the two breakpoint positions in the reference genome. This subclass extends the framework to include SV signal types that do not yet have a specific subclass implemented. For example, the a set of variants that are known to exist in the population can be included in the analysis of an individual or variants that are known to exist in a particular type of cancer can be included in the analysis of a tumor. This signal can be included in the analysis by expanding the edges of the predicted intervals to create breakpoint intervals, and encoding these intervals in BEDPE format. Each BEDPE entry is assumed to be a real breakpoint (`is_bp`), the variety is encoded in the auxiliary fields in BEDPE (`get_v`), and the intervals are directly encoded in BEDPE (`get_bpi`).

6.3 Results

6.3.1 Performance Comparisons

LUMPY's performance was compared to three other popular and actively maintained SV discovery packages: GASVPro [18], DELLY [19] and PINDEL [40]. These algorithms were selected due to their widespread use and inclusion in large-scale projects such as The Cancer Genome Atlas (GASVPro) and 1000 Genomes (DELLY and PINDEL). Moreover, PINDEL was one of the first published SV discovery tools, and GASVPro and DELLY both consider a secondary SV signal along with paired-end alignments (read-depth and split-read, respectively). Both GASVPro and DELLY have also demonstrated substantial improvement over other popular SV tools such as Breakdancer [16] and HYDRA [13].

Detection performance was measured using both simulated data and previously published Illumina sequencing data from the widely studied NA12878 CEPH individual. The first simulation measured each tool's basic detection capabilities in a prototypical scenario by simulating 2500 homozygous variants from various SV classes at random genomic locations. The second simulation assessed the power of each tool to detect 5516 known deletion variants present at varying allele frequencies within a mixed sample, as often occurs in heterogeneous tumors. Lastly, the analysis of SVs in the NA12878 genome assessed the performance of each tool on real data containing sequencing errors and other detection confounders that are difficult to simulate. In addition, the analysis of NA12878 (and her parents) measured the effect of considering either multiple samples or prior SV knowledge on LUMPY's performance. In each case, performance was measured in terms of sensitivity and false discovery rate (FDR) by comparing the predicted SV breakpoints to known breakpoints. Predictions that overlapped simulated variants were considered true positives, and all other predictions were considered false positives.

The simulated results were based on alignments generated by NOVOALIGN, and the NA12878 results were based on BWA alignments. In both cases, YAHA was used to generate split-read alignments. LUMPY was also tested using alternative read alignment pipelines using either BWA-backtrack or BWA-MEM for paired-end alignment, and BWA-MEM for split-read alignment (with roughly similar results; see Figure 6.2d and 6.2e).

Homozygous variants of different SV types

To assess the impact of coverage, SV type and SV size on algorithm performance, a set of sample genomes was simulated that included 2500 deletions, tandem duplications, inversions and translocations, randomly placed throughout the human genome (build 37). Variants were created with an in-house SV simulation tool (SVSIM) that made alterations to the reference genome [41]. For each SV type, the variant size ranged from 100bp to 10kb. Then the WGSIM read simulator [35] was used to sequence each simulated genome at 2X, 5X, 10X, 20X, and 50X haploid coverage.

LUMPY was consistently more sensitive than the other algorithms across nearly all coverage levels and SV types (Figure 6.2a). DELLY detected three more translocations than LUMPY at 20X coverage, at the expense of 93 more false positives. LUMPY and DELLY were the only algorithms that detected all variant types; GASVPro and PINDEL did not support detection of tandem duplications (tandem duplication support has since been added to PINDEL). LUMPY's superior sensitivity was most dramatic in lower coverage tests (<10X). For example, LUMPY detected 32.4% and 87.2% of all deletions at 2X and 5X coverage, respectively, whereas GASVPro detected 7.4% and 49.8%, DELLY detected 8.4% and 57.2%, and PINDEL detected 7.4% and 50%. At best, LUMPY was 35.5 times more sensitive than PINDEL for detecting translocations at 5X coverage (69.1% vs. 2%). At worst, LUMPY was 1.06 times more sensitive that DELLY for detecting translocations at 2X coverage (69.1% vs. 65.1%). At higher coverage (10-50X), LUMPY's sensitivity advantage persisted; it ranged from 88.8% to 99.6% across all SV types, whereas GASVPro ranged from 14.3% to 94.3%; DELLY ranged from 76.7% to 96.8%; and PINDEL ranged from 0.2% to 96.5% (excluding the SV types which GASVPro and PINDEL are incapable of detecting).

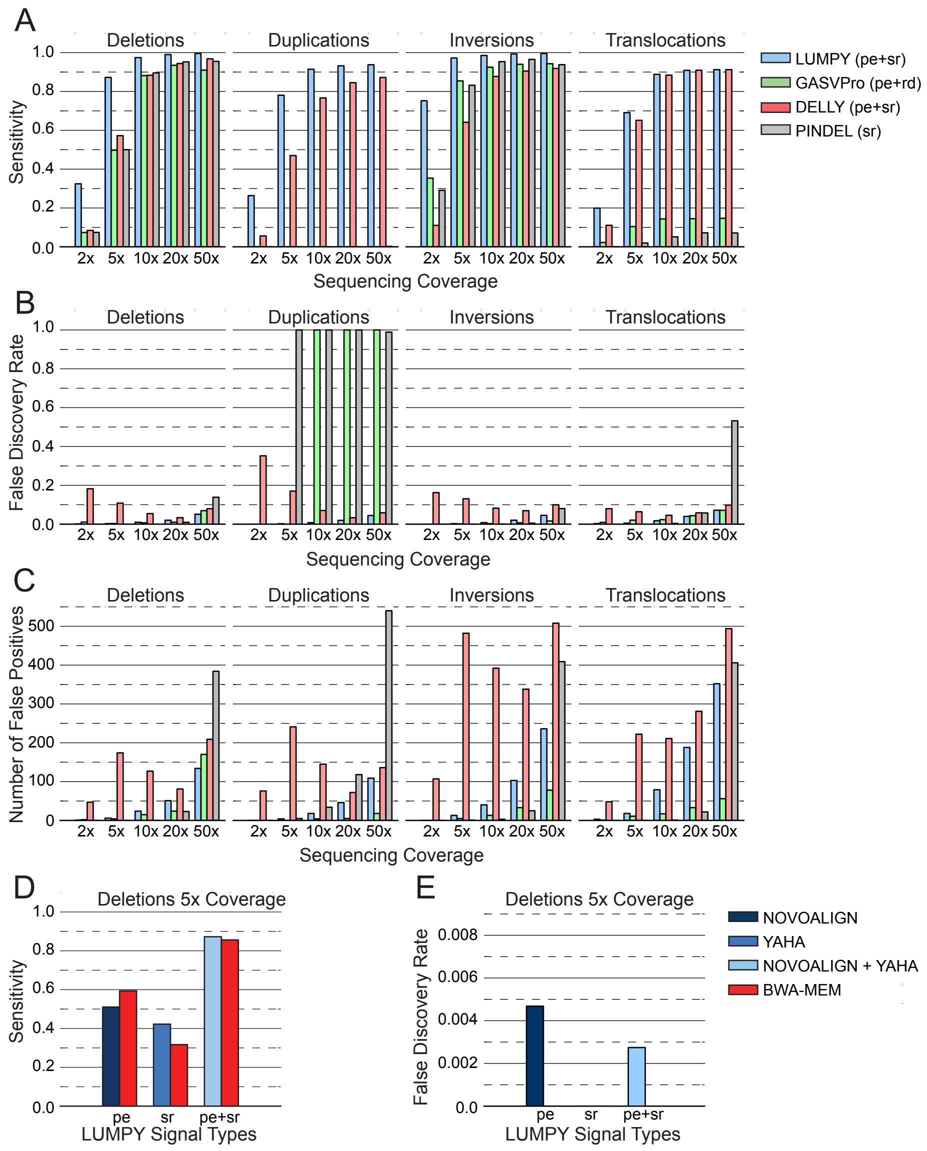


Figure 6.2: Sensitivity and accuracy for 2500 deletions, tandem duplications, inversions or translocations randomly embedded at different levels of sequence coverage. LUMPY and DELLY considered paired-end (pe) and split-read (sr) alignments, GASVPro considered paired-end alignments and read-depth (rd), and PINDEL considered split-read (sr) alignments. (A) Sensitivity. LUMPY was the most sensitive in most cases, and had a marked improvement at lower coverage levels. (B) False discovery rate (FDR). The FDR for LUMPY was low in all but the highest coverage cases. GASVPro and PINDEL do not support tandem duplications, but false calls were made in some cases which resulted in a 100% FDR. (C) The absolute number of false positive calls. LUMPY has a high number of false positives in some cases, but these are counterbalanced by a higher number of true positives (part a), resulting in a similar FDR (part b). The impact to LUMPY's sensitivity (D) and false discovery rate (E) for different sequence alignment strategies for the homozygous SV simulation presented in 5x deletion sections of (A), (B), and (C). Since BWA-MEM produces both paired-end (pe) and split-read (sr) alignment signals in a single alignment step, it serves as a basis of comparison to the NOVOALIGN (pe) and YAHA (sr) strategy that was used for the presented LUMPY results. Either alignment strategy is suitable and demonstrates the generality of the LUMPY framework for SV detection.

LUMPY's FDR remained low (less than 2%) in all but the highest coverage cases (Figure 6.2b), and there was only one instance where LUMPY's FDR was more than 2 percentage points higher than the best performing tool (GASVPro's FDR for inversions at 50X was 1.6% while LUMPY's was 4.5%). In general, the FDR for LUMPY, GASVPro, and PINDEL increased as coverage increased, ranging from 0% to 7.2% for LUMPY, 0% to 7.1% for GASVPro, and from 0% to 53.2% for PINDEL. In contrast, the FDR for DELLY dropped as coverage went from 2X to 20X, then increased again at 50X, ranging between 3.3% and 35.2%. These patterns suggest that coverage-based parameter tuning could be used to minimize FDR for all the tools. FDR calculations depend on the number of true positives, which vary widely across SV varieties (Figure 6.2c). In certain cases (e.g., translocations), LUMPY had a far higher absolute number of false positives, but these were counterbalanced by a much higher number of true positives. Alternatively, in cases where a specific SV type was not supported and no true positive calls were made (i.e., GASVPro and PINDEL for tandem duplications), the FDR reached 100%.

Heterogeneous tumor simulation

Variant detection is especially challenging in tumor studies because biopsied samples generally include a mixture of abnormal and normal tissue, and because many tumor samples are composed of multiple clonal lineages defined by distinct somatic mutations. To assess the performance of LUMPY in this more realistic scenario, where increased sensitivity is crucial, heterogeneous samples were simulated by pooling reads from an abnormal genome and a normal genome at varying ratios. The source of the simulated abnormal genome was the human reference genome (build 37) modified (using SVSIM) with 5516 non-overlapping deletions identified by the 1000 Genomes Project [42], and an unmodified human reference genome was used to simulate the normal genome. As above, each genome was sequenced using WGSIM, and the reads from the two genomes were combined in varying proportions to create a single heterogeneous sample. The ratio of the reads from the abnormal genome (SV allele frequency) varied between 5% and 50%, and the total coverage ranged from 10X and 80X. For example, to simulate a sample with a 5% SV allele frequency at 10X coverage, the abnormal genome was sequenced at 0.5X coverage and the normal genome at 9.5X coverage: when combined, the two sets of reads represented a single heterogeneous sample sequenced at 10X coverage.

LUMPY was more sensitive than GASVPro, DELLY, and PINDEL in all cases, especially when the coverage of the abnormal genome was low owing to either lower coverage, low SV allele frequency, or both (Figure 6.3a). For example, at 10X coverage and 20% SV allele frequency LUMPY detected 31% of the SVs, whereas GASVPro, DELLY, and PINDEL detected only 9.2%, 10.9%, and 6.2% of the SVs, respectively. This represented a 2.9-fold increase in sensitivity over the next best method. In general, to achieve the same level of sensitivity, GASVPro, DELLY and PINDEL required roughly twice as much evidence as LUMPY (by either increased coverage or SV allele frequency). For example, at 20X coverage LUMPY detected 6.2% of variants present at 5% SV allele frequency, whereas GASVPro, DELLY, and PINDEL required 10% SV allele frequency to achieve similar sensitivity (10.3%, 8.8%, and 5.5%, respectively). This trend was also apparent across SV varieties in the previous homozygous test (Figure 6.2a).

The FDR for LUMPY was lower than all other tools in all cases (Figure 6.3b), ranging from 0% to 3.4%. For GASVPro, DELLY, and PINDEL, the FDR was particularly high when SV allele frequency was low. For example, at 10X coverage the FDR for GASV was 5.8 times higher at 5% SV allele frequency than at 50% frequency, DELLY FDR was 3.5 times higher, and PINDEL FDR was 111.9 times higher. At 20X coverage these differences are 4.1, 10.9, and 21.5 times higher at 5% frequency than at 50% frequency for GASVPro, DELLY, and PINDEL, respectively. This was in contrast to LUMPY, where modest coverage-associated increases to FDR can likely be managed via parameter tuning, without significantly decreasing sensitivity.

SV detection in the NA12878 genome

Although it is difficult to precisely measure the sensitivity and accuracy of SV predictions made from a real data set, it is important to evaluate each tool's performance when confronted with real data containing artifacts that are not easily captured by simulations (e.g., PCR artifacts, chimeric molecules, reads from poorly assembled genomic regions, etc.). This experiment compared SV detection performance in the NA12878 individual by analyzing the Illumina Platinum Genomes dataset, which represents 50X coverage of the NA12878 genome (European Nucleotide Archives; ERA172924). This data set was also subsampled to 5x coverage to assess SV detection in low coverage scenarios.

To estimate sensitivity and FDR, predictions made by each tool were compared to two truth sets: the first set was based on 3376 validated, non-overlapping deletions from the 1000 Genomes project [42], and the second set included the same 3376 validated deletions as well as 2128 additional deletion predictions made by at least two tools using the 50X dataset (5504 total deletions). The rationale for two truth sets is that, although the 1000 Genomes callset is the most comprehensive set of deletions for NA12878, it still represents only a subset of the actual deletions in that individual's genome. This study had the benefit of higher quality sequencing data, longer reads and improved SV detection tools, and thus was likely to discover novel deletions that were missed by Mills et al.[42]. Furthermore, since PINDEL was one of the tools used to generate the 1000 Genomes callset[42], it was biased against predictions made by LUMPY, GASVPro and DELLY. Therefore, although some of the predictions included in the larger truth set may not reflect genuine SVs, these calls nonetheless represented real differences between the sequencing data and the reference genome and, as such, provided a more realistic estimate of the FDR for each tool.

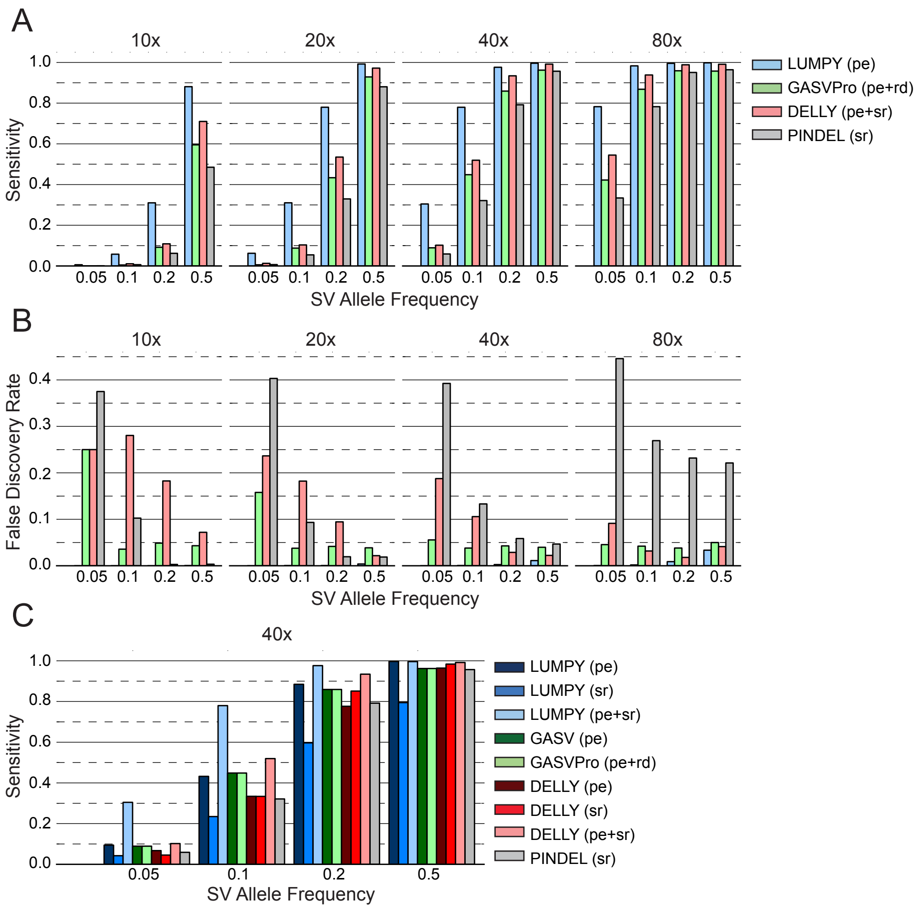


Figure 6.3: Performance comparison using known deletions at varying intra-sample allele frequencies. To measure SV detection sensitivity and accuracy in the case of a heterogeneous tumor sample, 5516 non-overlapping deletions identified by the 1000 Genomes Project were embedded in an ”abnormal” genome prior to read simulation. Simulated reads from the abnormal genome and the unaltered reference genome were mixed in varying proportions to obtain simulated datasets with different SV allele frequencies. Sequencing coverage levels are shown above each plot, and SV allele frequencies are shown beneath each plot. (a) SV detection sensitivity for the four tools. LUMPY and DELLY considered paired-end (pe) and split-read (sr) alignments, GASVPro considered paired-end alignments and read-depth (rd), and PINDEL considered split-read (sr) alignments. In all cases, LUMPY was more sensitive than GASVPro, DELLY, and PINDEL, and showed a marked improvement when the coverage of the abnormal genome was low due either to low sequence coverage or low SV allele frequency. In general, to achieve the same level of sensitivity as LUMPY, the other tools required twice as much evidence from the abnormal genome. (b) The false discovery rate (FDR) for each tool. The FDR for LUMPY was better than all other tools in all cases, with a notable improvement when SV allele frequency was low. (c) The change in sensitivity when considering two SV detection signals versus a single signal alone is shown for the three tools at 40x coverage and at different SV allele frequencies. At low SV allele frequencies (e.g., 5%), LUMPY’s use of two signals (i.e., pe+sr) has a super-additive effect on sensitivity relative to either signal alone (i.e., pe or sr), whereas the sensitivity of GASVPro and DELLY was either unchanged or modestly improved with one signal versus two.

A unique strength of LUMPY relative to other tools is the ability to consider different types of evidence from multiple samples. To demonstrate this capability, results for three different LUMPY configurations were included: (1) the standard configuration of read-pair and split-read signals from NA12878; (2) read-pair and split-read signals from NA12878, as well as prior knowledge (using LUMPY's generic evidence module) of deletions discovered by the 1000 Genomes Project using low coverage whole genome sequencing (phase 1, release version 3); (3) read-pair and split-read signals from both NA12878 and her parents (NA12891 and NA12892). The last two calling strategies are specific to LUMPY and are intended to demonstrate both the ability and the benefit of including data from different samples and from prior results.

At 5X coverage (Figure 6.4a), LUMPY was more sensitive than both GASVPro and PINDEL (17.6% versus 9.5% and 16.1% for the first truth set and 13.5% versus 7.6% and 10.8% for the second truth set) and had an either equivalent or better FDR (33.7% versus 56.1% and 33.4% for the first truth set and 18.9% versus 43.9% and 28.5% for the second). While DELLY was more sensitive than LUMPY (19.3% for the first truth set and 16.1% for the second), it was at the expense of at least 0.5-fold higher FDR (51.5% and 35.1%). However, when LUMPY was provided with priors from the 1000 Genomes low coverage deletion calls, sensitivity was increased to 22.5% with a negligible effect on false positives (leading to a lower FDR). Sensitivity was further improved to 27.0% when LUMPY performed simultaneous variant calling on NA1287 and her parents, with a similarly small effect on FDR, which clearly demonstrated the benefit of pooled variant calling on genetically related samples. Overall, the effect of the second truth set at lower coverage was similar between LUMPY, GASVPro, and DELLY with FDR decreasing by 14.8, 12.2, and 16.4 percentage points, respectively. The effect on PINDEL was the smallest, with FDR decreasing by only 5 percentage points; however, this smaller effect was expected considered that PINDEL was used in part to create the baseline truth set [42].

At 50X coverage (Figure 6.4b) and considering the first truth set, LUMPY, PINDEL and DELLY had similar sensitivity (62.1%, 63% and 61.3%, respectively), and all outperformed GASVPro (55.1%). Owing to the incompleteness of the first truth set, all of the tools had high FDR. LUMPY performed the best at 54%, followed by GASVPro at 70.9%, PINDEL at 72%, and DELLY at 84.8%. When considering the second truth set, the effect on performance was dramatic for all the tools except PINDEL (as expected). LUMPY had the highest sensitivity at 73.3%, followed by DELLY at 71.3%, GASVPro at 61.9%, and PINDEL where the sensitivity actually dropped to 51.8% (due to the increased size of the second truth set). LUMPY also had the lowest FDR at 14.8%, followed by GASVPro at 42%, PINDEL at 63.1%, and DELLY at 71.5%. As expected given the substantially higher coverage, the inclusion of either priors from known SVs or the parental genomes had little effect on LUMPY's sensitivity. Overall, these results demonstrate that LUMPY provides substantial improvements in discovery sensitivity over existing methods while also maintaining

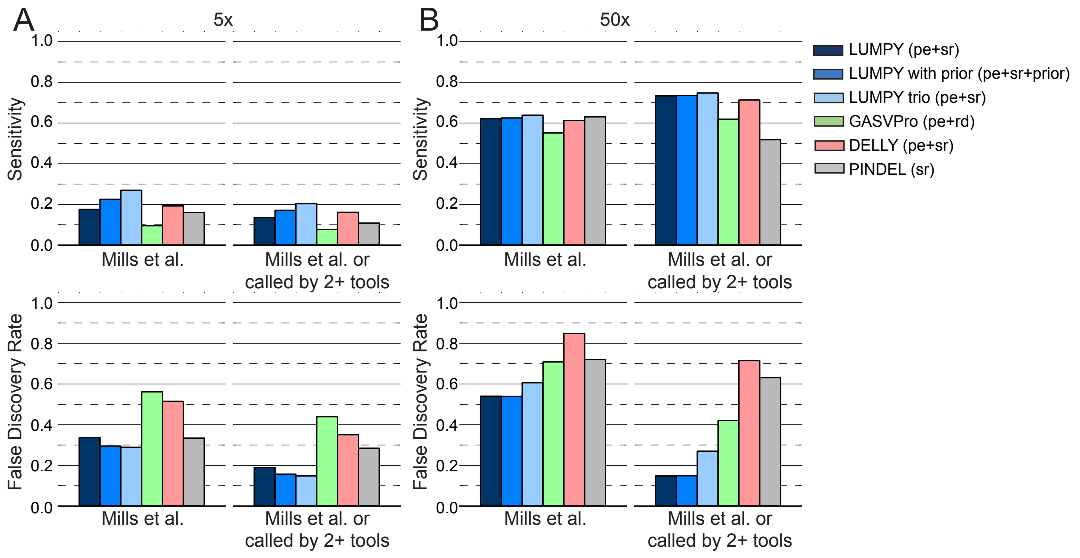


Figure 6.4: Performance comparison of deletion detection in high and low coverage Illumina sequencing data from the NA12878 individual. To measure each tool's performance on data that is representative of a typical experimental data set, previously published Illumina sequencing data from the NA12878 individual that was aligned using BWA was considered. LUMPY considered paired-end (pe) and split read (sr) alignments for NA12878, LUMPY prior considered paired-end alignments and split read alignments for NA12878 and 1000 Genomes variants as prior evidence (prior), LUMPY trio considered paired-end alignments and split read alignments for NA12878 and her parents NA12891 and NA12892. DELLY considered paired-end (pe) and split-read (sr) alignments, GASVPro considered paired-end alignments and read-depth (rd), and PINDEL considered split-read (sr) alignments. Sensitivity and FDR was estimated using non-overlapping validated deletions from Mills et al. [XX] as well as predictions made by at least two tools. (a) SV detection sensitivity and FDR for the four tools on a 5X coverage subsample of the original data. LUMPY was more sensitive than both GASVPro and PINDEL and had an either equivalent or better FDR. DELLY was slightly more sensitive than LUMPY, but also had a much higher FDR. The inclusion of prior evidence or parental genomes improves sensitivity without increasing false positives. (b) SV detection sensitivity and FDR for the four tools on the original 50X coverage data. LUMPY, DELLY, and PINDEL had similar sensitivity in the Mills et al. case, and in the extended truth set LUMPY and DELLY had the highest sensitivity. LUMPY had lower FDR than all other tools in both cases. The effect of prior evidence and parental genomes is reduced in higher coverage data.

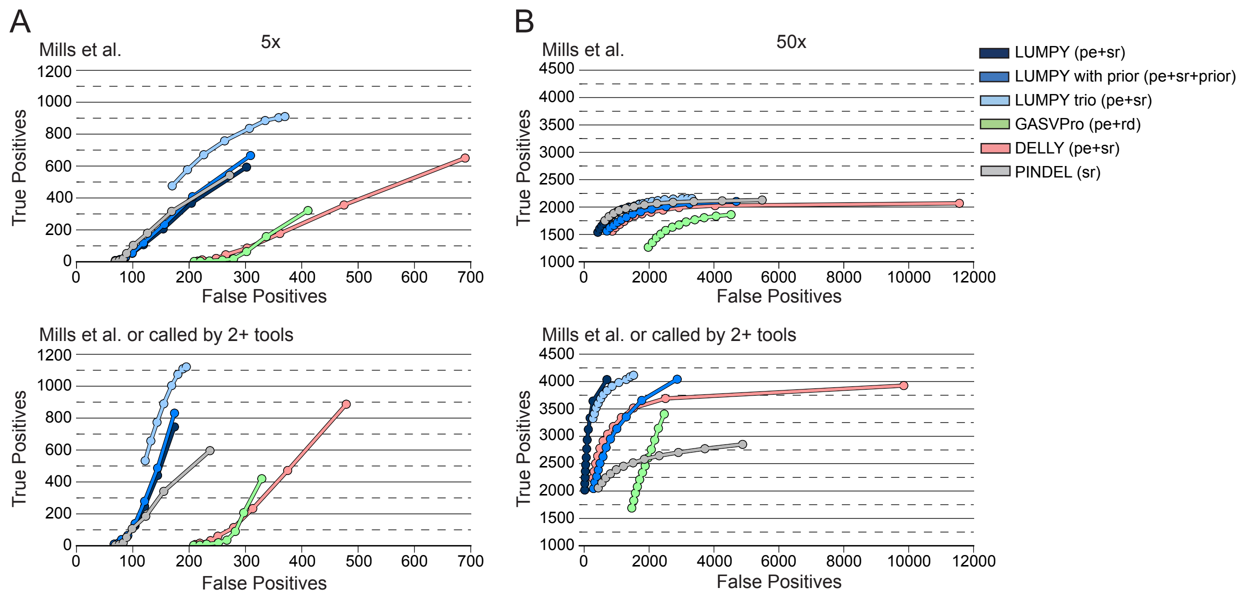


Figure 6.5: ROC curves comparing deletion prediction performance in the NA12878 individual. ROC curves are shown comparing the correctly predicted deletions made by LUMPY, GASVPro, DELLY, and PINDEL to the deletions in the standard Mill et al truth set, as well as the extended truth set which also includes the deletion predictions identified by two or more tools. As in Figure 6.4, prediction performance was measured with both 5X average genome coverage (A) and 50X average genome coverage (B). Each tool was given the same set of input alignments in BAM format and the SV calls made by each tool were compared to the truth sets using identical criteria (see Methods). Each point on a given tool's ROC curve represents a minimum evidence support threshold ranging from 4 to 11, inclusive for 5X coverage and 4 to 15, inclusive for 50X coverage. The curves are colored following the same convention described in Figure 6.4.

equivalent or lower false discovery rates.

Importantly, while the comparisons presented in Figure 6.4 were based upon the choice of a single detection threshold chosen for each tool, LUMPY's detection accuracy remained superior across a broad spectrum of thresholds (Figure 6.5), indicating that the framework itself, not arbitrary parameter choices, underlie LUMPY's increased performance.

6.3.2 Benefits of Integrating All Signals for SV Discovery

LUMPY's superior sensitivity in these performance tests is a direct consequence of the fact that it simultaneously integrates multiple SV detection signals during SV discovery. The benefits of this approach are clear from the super-additive effect of combining read-pair and split-read signals within the LUMPY framework, relative to using either signal alone (Figure 6.3c). In contrast, although other tools such as GASVPro [18], DELLY [19] and Genome STRiP [36] also exploit multiple SV detection signals, they first use one signal (i.e., read-pair) to drive discovery and then refine and genotype candidates with a second signal (i.e., split-read or read-depth). An intrinsic limitation of stepwise integration is that other available signals cannot increase the number of true positive SV calls beyond those candidates identified by the signal used for initial discovery. Consistent with this interpretation, inclusion of a second SV detection signal has little to no effect on DELLY's or GASVPro's sensitivity (Figure 6.3c) relative to using the primary read-pair signal alone.

6.3.3 Comparison Details

Both simulated and real datasets were used to compare the sensitivity and false discovery rate of LUMPY to other SV detection algorithms (GASVPro, DELLY, and PINDEL). Two types of simulations were performed: one in which homozygous variants of diverse varieties were introduced at random positions throughout the reference genome, and another in which a heterogeneous tumor sample was simulated by mixing reads from a modified "abnormal" human reference genome (containing 1000 Genomes deletions) and a unmodified "normal" human reference genome in varying proportions. The publicly available Illumina sequencing data of the NA12878, NA12891, and NA12892 individuals was also used. Two scenarios were considered: the original 50X coverage files, and 5X subsamples of the original data sets.

In the case of the homozygous simulation, SVSIM was used to create new versions of the human reference genome (build 37) containing 2500 simulated variants of each variety. For deletions, tandem duplications and inversions 2500 non-overlapping variants ranging from 100 bp to 10000 bp in size were randomly placed in the genome. To simulate translocations, 2500 non-overlapping inter-chromosomal regions of 1000 bp, derived from random donor sites in the reference genome were randomly inserted. Although the true variant variety in this case is actually an insertion, the inserted segment exceeds the insert size of the sequencing library as well as the read length, and thus the breakpoints formed by such insertions accurately simulate a translocation. Each simulated genome was sampled to 40X, 20X, 10X, 5X, and 2X coverage.

To simulate a heterogeneous tumor sample, simulated reads from both a modified and unmodified version of the human reference genome (build 37) were combined. The modified genome was created using SVSIM, and included 5516 non-overlapping deletions identified by the 1000 Genomes Project. Each simulation combined reads from both the modified and unmodified genomes in varying proportions. The proportion of reads that were derived from the modified genome are referred to as the SV allele frequency. The simulated SV allele frequencies were 5%, 10%, 20% and 50%, and the simulated coverages were 10X, 20X, 40X, and 80X. For example, in the simulation with 5% SV allele frequency and 10X coverage, the modified genome was sampled at 0.5X coverage and the unmodified genome was sampled at 9.5X coverage. The two sets of reads were then pooled into a single 10X coverage sample.

For all simulations, WGSIM was used to sample paired-end reads with a 150 bp read length, a 500 bp mean outer distance with a 50 bp standard deviation, and default error rate settings. Paired-end reads were mapped to the reference genome with NOVOALIGN version V2.07.08, using the random repeat reporting and allowing only one alignment per read. From the NOVOALIGN output, all soft-clipped ($\leq$20 bp clipped length) and unmapped reads were realigned with the split-read aligner YAHA using a word length of 11 and a minimum match of 15. The NOVOALIGN output was used as input to DELLY ,GASVPro, and PINDEL, and both NOVOALIGN and YAHA output were used as input to LUMPY. In all algorithms, the minimum evidence threshold was four. For LUMPY, the tuning parameters were:

- `min_non_overlap` was set to 150
- `discordant_z` was set to 4
- `back_distance` was set to 20
- `weight` was set to 1
- `min_mapping_threshold` was set to 1

For GASVPro the parameters were:

- `LIBRARY_SEPARATED` was set to all
- `CUTOFF_LMINLMAX` was set to SD=4
- `WRITE_CONCORDANT` was set to true
- `WRITE_LOWQ` was set to true

For DELLY the parameters were:

- `map-qual` was set to 1
- the `inc-map` flag was set

DELLY paired-end (pe) and split-read (sr) calls were combined into a single paired-end and split-read (pe+sr) call set by taking the union of the two sets where the split-read call was retained when a call was common to both sets. For PINDEL the parameters were:

- `minimum_support_for_event` was set to 4
- all chromosomes were considered
- `report_interchromosomal_events` was set to true

For the real data, LUMPY, GASVPro, DELLY, and PINDEL considered Illumina sequencing of the NA12878 individual. The LUMPY trio results also considered sequencing data from that individual's parents, NA12891 and NA12892. All sequencing samples were retrieved from the European Nucleotide Archives (submission ERA172924), and were previously aligned using BWA. Soft-clipped (20 bp clipped length) and unmapped reads were realigned with the split-read aligner YAHA using a word length of 11 and a minimum match of 15. The LUMPY prior results also considered all 1000 Genomes variant calls using the generic evidence module. The original sequencing files were at 50X coverage and were used in the 50X experiments. The 5X experiments considered sequencing files that were created by subsamples 10% of the original paired-end alignments. For all the tools, only deletion predictions on chromosomes 1 though X were considered. Each tool was run with the same options that were used in the simulation experiments, except for the minimum mapping quality for LUMPY, GASVPro, and PINDEL was increased to 10. Since PINDEL uses paired-end reads differently than the other tools, the default mapping quality of 20 was used. Each call required support of at least four. In the LUMPY trio result, a call had to have support of four from at least one individual (NA12878, NA12891, or NA12892) and at least one piece of support from NA12878. The weight for the 1000 Genomes variant calls in the LUMPY prior result was set to 2. For the identification of the truth sets, the Mills et al. study validated 14012 deletions in NA12878 across 11 independent laboratories. Once duplicate predictions were removed, the first truth set contained 3376 non-overlapping deletions. Between the four tools considered here, 4027 predictions were made by at least 2 tools (2035 were made by all four tools), and 2128 of those merged predictions were not observed by Mills et al. The 3376 Mills et al. validated deletions were combined with the 2128 additional predictions to produce a second, more comprehensive, truth set containing 5504 deletion calls.

The SV breakpoints predicted by each algorithm were compared to the known variants. A true positive was a predicted variant call that intersected within 50 bp of both ends of the simulated breakpoints in the reference genome. All other predictions were considered to be false positives. Since the output of DELLY is a single interval, 100 bp regions flanking the ends of the predicted interval were taken as the predicted breakpoint.

6.4 Conclusion

LUMPY is a general probabilistic framework for SV discovery that is more sensitive than existing discovery tools across all SV types and coverage levels, and in both real and simulated human genome datasets. LUMPY's high sensitivity is a direct consequence of combining multiple SV detection signals. LUMPY integrates disparate signals by converting them to a common format in which the two predicted breakpoint intervals in the reference genome are represented as paired probability distributions. SV prediction operations are then performed at this higher level. This novel approach has the key advantage that any SV detection signal can be integrated into the framework so long as breakpoint probabilities can be assigned to each base pair in a candidate breakpoint region. Potential detection signals include paired-end and split-read alignments, alignments from assembled contigs, raw read-depth measurements, or CNVs detected by segmentation of read-depth, array comparative genomic hybridization, or SNP array data. As demonstrated in the analysis of the NA12878 genome, inclusion of previously discovered SVs as priors can significantly enhance SV discovery sensitivity, which is an example of the flexibility and generality of the framework. As sequencing technologies and SV detection strategies evolve, new sources of evidence can be easily incorporated without modifying the underlying logic of the SV detection algorithm itself; the sole requirement is the development of a new module that maps the SV detection signal to a paired probability distribution. Importantly, LUMPY's increased sensitivity does not come at the cost of excessive spurious SV predictions. LUMPY therefore represents an important technological advance, especially in the context of cancer genomics where sensitivity is crucial for identifying low frequency variants within heterogeneous tumor samples.

Chapter 7

Conclusion

This thesis presented three algorithms that have significantly advanced the fields of bioinformatics and computational biology. The BITS algorithm introduced a new way to efficiently divide an interval set into subsets that begin before and end after a particular interval, which enabled an optimal sequential algorithm to perform 12x faster than the next best solution and a parallel algorithm to perform 3,414x faster. The slice-then-sweep algorithm carried this insight forward into the consideration of many interval sets, yielding a sequential speedup of 2x and a parallel speedup of up to 19x. LUMPY generalized the concept of a genomic breakpoint to allow for a significant expansion in the number of signals that can be considered in SV detection, which resulted in a dramatic improvement to sensitivity in both normal human tissue and heterogeneous samples such as tumor genomes. While these contributions empower researchers to consider the vast amounts of data that will be produced in the coming years, the pace of advancement requires continuous innovation. This thesis concludes with a number of specific improvements and extensions to this work, as well as more general thoughts on future directions.

Given the efficiency with which the BITS algorithm counts intersections, it is also well suited to other genomic analyses including RNA-seq transcript quantification, ChIP-seq peak detection, and searches for copy-number and structural variation. Moreover, the functional and regulatory data produced by projects such as ENCODE have led to new approaches [43] for measuring relationships among genomic features. Many of these new tests are based on counting the occurrence of particular relationships, and it is likely that some can be efficiently solved using a BITS-style algorithm. For example, implementing the absolute distance test, which measures the significance of the distance between genomic intervals, would require only two extra comparisons beyond the standard BITS method. Other tests, such as the Jaccard test, which focuses on the total size of the intersecting set, are substantively different than BITS but are still good candidates for parallelization. Since determining which test is most insightful depends on many factors, an efficient solution that calculates the results of many tests will be the most useful. For example, using only the intersection-based test would fail to characterize the significance of the relationship between two features that are in close proximity but do not intersect, such promoters and genes.

An important generalization of the N-way intersection problem is to find intersections occurring among at least k of N interval sets. While the slice-then-sweep algorithm does not solve the "k of N" problem, it may be possible to extend the algorithm by either merging level-one slices for some subset of the N set or by considering inverted slices (formed by space between intervals). The outstanding issue with these solutions would be to prevent an exponential number of comparisons between slices. Another important consideration for future research is to minimize the memory footprint of the algorithm in order to enable scaling to thousands of genomic interval sets. Recent advances such as Tabix [44] allow random access into overlapping intervals within compressed genomic interval files. Modifications to this format such as Grabix [45] allow random access to arbitrary records in such files and therefore permit slicing interval sets without loading them into memory. Extending the slice-then-sweep algorithm to leverage such functionality could greatly boost scalability and enable researches to discover new relationships between genomic features that were previously thought to be unrelated.

The most obvious improvement to LUMPY would be the inclusion of read-depth information for duplication and deletion variants. At present, this can be accomplished by incorporating the result of an external read-depth analysis tool through LUMPY's generic module; however, more significant improvements should be possible using raw read-depth data. Second, for applications that require ultra-sensitive detection of known structural variants such as low-coverage population scale sequencing LUMPY could be packaged with dataset priors reflecting the positions and allele frequencies of previously identified SVs. While this is feasible using the existing LUMPY framework, substantial improvements to sensitivity may require more comprehensive and accurate SV catalogs than are currently available. Finally, a major challenge for SV detection is distinguishing bona fide variants from false positives caused by alignment artifacts and other sources of error. By using machine learning to train on a set of known variants, it should be possible to derive a probabilistic measure of variant confidence that is based not only on the number of clustered reads, but also on the shape of the final integrated probability distribution. Alternatively, knowledge of the shape of "true" breakpoint probability distributions could potentially be used as an objective function during read clustering. In a more general sense, LUMPY's approach for integrating SV detection signals - in essence, performing genome interval comparison operations using probability distributions rather than "flat" features - could be useful for any application that involves comparison of genomic features whose exact coordinates are unknown, and whose positional uncertainty can be represented rationally in the form of a probability distribution. Rapid and efficient probabilistic comparisons could be enabled through extensions to existing interval-based software such as BEDTools [6, 28].

LUMPY's performance makes it a good candidate for inclusion in clinical applications. Although it was not part of the formal comparison, LUMPY made SV predictions significantly faster than other widely used tools. When considering high-coverage data sets, LUMPY returned results in less than an hour when other tools required days and in some cases over a week to complete. This difference is critical in many time-sensitive clinical settings, such as treatment decisions for high-grade cancers and genetic diagnostics in neonatal intensive care units. In these cases, SV predictions would help determine the course of treatment, but doctors cannot wait days to make a decision. To address this issue, LUMPY has been included in a larger bioinformatics pipeline that aims to characterize and prioritizing genetic variation in under 24-hours. By moving raw sequencing data to a fully characterized genome in a relevant time line, the pipeline aims to remove a significant barrier to the clinical adoption of genome sequencing technologies.

More broadly, the current trend in genomics is to gain analytic power by drastically increasing sample sizes. Projects that include thousands of individuals have already been completed, and new projects will attempt to increase that number to a million. Unfortunately, the tools required to analyze these future data sets are not up to the task. While the work presented here is a step in the right direction, the need for more scalable algorithms and architectures remains. A number of different methods have been developed to handle large volumes of information. In particular, NoSQL solutions like BigTable [46] have had success in some problem domains, but the adoption in genomics remains small. This is partly due to the systems that these solutions run on and the sensitivity of the data. Many researchers are weary of placing their data, which can contain identifiable genetic information, on shared resources like Amazon cloud. However, this resistance is likely to be temporary given the direction of the field and advances in data security. The National Institute of Health (NIH) has recognized the waste associated with replicating data sets on individual research clusters. Going forward, the model will be to bring algorithms to the data, and not data to the algorithms.

A more fundamental issue is the extent to which these new architectures aid the types of analysis that are on the horizon. NoSQL refers to a class of key-value based data stores that are highly flexible and optimized for many simple operations. For example, Facebook is using HBase [47] (a popular open-source clone of Google's BigTable) to manage a user messaging service that currently handles 500 million transactions per day. While this use-case clearly demonstrates scalability, it does not demonstrate applicability. NoSQL queries return a set of rows based on a key. It is not clear how this operation can be used to support the intersection of many interval sets, or even the efficient intersection of a single large interval set. One solution is to place intervals in bins (similar to an R-Tree), and then let the bin ID be the key and the set of intervals in a bin be the value. To find an intersection for a given query interval, the IDs of the bins that intersect the query are calculated, and the intervals in those bins are retrieved. While this technique does reduce the number of intervals that are considered, since the number and size of the bins is fixed, the number of intervals in each bin grows linearly with the number of intervals in the set. Clearly a new method that more naturally supports intervals is required in order to leverage the scalability that NoSQL data stores provide. The main challenge for this method will be to address the two most typical access patterns: scan all intervals across a given sample, and scan all samples containing a given interval. This will be particularly difficult since most data stores are optimized to handle only one of these access patterns, typically at the expense of the other.

Furthermore, the technical barriers associated with large-scale genomic analysis must be reduced. Currently, researchers analyze data sets using a collection of command line tools weaved together with Unix utilities like `grep` and `awk`. Similar to how SQL and newer query languages like HIVE and PIG have made large-scale data analytics possible for a wider section of less technical users, an interval-based genome query language would make the large volumes of published data more accessible to biologists, and allow them to concentrate on hypotheses generation and verification. There is some early work in this area, but these solutions are limited to either work flow tools [4] or domain-specific languages that focus on only one type of analysis [48]. To be successful a general-purpose genomic analysis language must be both powerful and usable, which tend to be competing goals. Given its success, SQL should be a model for striking this balance.

Low-cost, high-throughput genome sequencing will have a significant impact on the future of research and medicine. The challenge will be to provide access to such a vast amount of data while simplifying the process so that users with less technical training (e.g., clinicians, traditional biologists) can also leverage the data. One step that has already been taken is the development of a strategy that will allow analysis to be performed on the same systems that house the data (e.g., Amazon cloud). By reducing the burden associated with collecting disparate data sets and nearly eliminating the up-front storage and compute costs, large-scale analysis will become possible for even the smallest operations. While this is a major component of the scalability and accessibility solution, it only addresses the physical requirements. The appropriate software systems must also be deployed to ensure that data can be accessed efficiently and easily. An interval-based access system and an intuitive genomic query language can ensure that users are able to quickly find the data they need without the technical barriers currently associated with genomic analysis. Exactly how to achieve either of these solutions remains open, and should be at the forefront of computer science research.

Bibliography

[1] J.D. Watson et al. *Molecular biology of the gene.* Cold Spring Harbor Laboratory Press, sixth edition, 2007.

[2] International Human Genome Sequencing Consortium. Initial sequencing and analysis of the human genome. *Nature*, 209:860–921, 2001.

[3] W.J. Kent et al. The human genome browser at UCSC. *Genome Research*, 12(6):996–1006, 2002.

[4] B. Giardine et al. Galaxy: a platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451–1455, Oct 2005.

[5] H. Li et al. The sequence alignment/map (SAM) format and SAMtools. *Bioinformatics*, 25:2078–2049, 2009.

[6] A.R. Quinlan and I.M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–842, 2011.

[7] NVIDIA Corporation. NVIDIA CUDA C programming guide version 5.5, 2014.

[8] The ENCODE Project Consortium. An integrated encyclopedia of dna elements in the human genome. *Nature*, 489:57–74, 2012.

[9] B.E. Bernstein et al. The NIH roadmap epigenomics mapping consortium. *Nature Biotechnology*, 28(10):1045–8, 2010.

[10] A.R. Quinlan and I.M. Hall. BEDTools: a flexible suite of utilities for comparing genomic features. *Bioinformatics*, 26(6):841–42, 2010.

[11] A.V. Alekseyenko and C.J. Lee. Nested containment list (NCList): a new algorithm for accelerating interval query of genome alignment and interval databases. *Bioinformatics*, 23(11):1386–1393, 2007.

[12] J.E. Richardson. fjoin: simple and efficient computation of feature overlaps. *Journal of Computational Biology*, 13:1457–1464, 2006.

[13] A.R. Quinlan et al. Genome-wide mapping and assembly of structural variant breakpoints in the mouse genome. *Genome Research*, 20(5):623–35, 2010.

[14] S. Neph et al. BEDOPS: High performance genomic feature operations. *Bioinformatics*, 28:1919–1920, 2012.

[15] M. McKenney and T. McGuire. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 392–395, New York, NY, USA, 2009. ACM.

[16] K. Chen et al. BreakDancer: an algorithm for high-resolution mapping of genomic structural variation. *Nature Methods*, 6(9):677–81, 2009.

[17] S.S. Sindi et al. A geometric approach for classification and comparison of structural variants. *Bioinformatics*, 25(12):i222–30, 2009.

[18] S.S. Sindi et al. An integrative probabilistic model for identification of structural variation in sequencing data. *Genome Biology*, 13(3):R22, 2012.

[19] T. Rausch et al. DELLY: structural variant discovery by integrated paired-end and split-read analysis. *Bioinformatics*, 28(18):i333–9, 2012.

[20] J. Mirsa and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[21] D. Kirk and W. Hwu. *Programming Massively Parallel Processors: A Hands-On Approach.* Elsevier, 2010.

[22] D. Merrill and A. Grimshaw. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(3):245–272, 2011.

[23] N. Satish et al. Designing efficient sorting algorithms for manycore GPUs. In *International Symposium on Parallel and Distributed Processing, 2009*, IPDPS '09, pages 1–10. IEEE, 2009.

[24] The 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–73, 2010.

[25] S. Tzeng and L.Y. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games*, I3D '08, pages 79–87, New York, NY, USA, 2008. ACM.

[26] ENCODE Project Consortium. Identification and analysis of functional elements in 1% of the human genome by the ENCODE pilot project. *Nature*, 447(7146):799–816, 2007.

[27] M.B. Gerstein et al. Integrative analysis of the *Caenorhabditis elegans* genome by the modENCODE project. *Science*, 330(6012):1775–1787, 2010.

[28] R.M. Layer et al. Binary interval search: a scalable algorithm for counting interval intersections. *Bioinformatics*, 29(1):1–7, 2013.

[29] J.L. Bentley and T.A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–7, 1979.

[30] M.T. Maurano et al. Systematic localization of common disease-associated variation in regulatory DNA. *Science*, 337(6099):1190–5, 2012.

[31] ENCODE Project Consortium et al. A user's guide to the encyclopedia of DNA elements (ENCODE). *PLoS Biology*, 9(4):e1001046, 2011.

[32] C. Hercus. Novoalign. http://www.novocraft.com, 2014. [Online; accessed 21-January-2014].

[33] G.G. Faust and I.M. Hall. YAHA: fast and flexible long-read alignment with optimal breakpoint detection. *Bioinformatics*, 28(19):2417–24, 2012.

[34] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 26(5):589–95, 2010.

[35] H. Li. WGSIM. https://github.com/lh3/wgsim, 2014. [Online; accessed 21-January-2014].

[36] R.E. Handsaker et al. Discovery and genotyping of genome structural polymorphism by sequencing on a population scale. *Nature Genetics*, 43(3):269–76, 2011.

[37] M.A. DePristo et al. A framework for variation discovery and genotyping using next-generation dna sequencing data. *Nature Genetics*, 43(5):491–8, 2011.

[38] F. Hormozdiari et al. Simultaneous structural variation discovery among multiple paired-end sequenced genomes. *Genome Research*, 21(12):2203–12, 2011.

[39] A.R. Quinlan et al. Genome sequencing of mouse induced pluripotent stem cells reveals retroelement stability and infrequent dna rearrangement during reprogramming. *Cell Stem Cell*, 9(4):366–73, 2011.

[40] K. Ye et al. Pindel: a pattern growth approach to detect break points of large deletions and medium sized insertions from paired-end short reads. *Bioinformatics*, 25(21):2865–71, 2009.

[41] G. Faust and Hall I.M. SVSIM. Unpublished, 2014.

[42] R.E. Mills et al. Mapping copy number variation by population-scale genome sequencing. *Nature*, 470(7332):59–65, 2011.

[43] A. Favorov et al. Exploring massive, genome scale datasets with the GenometriCorr package. *PLoS Comput Biol*, 8(5), 2012.

[44] H. Li. Tabix: Fast retrieval of sequence features from generic TAB-delimited files. *Bioinformatics*, 27:718–719, 2011.

[45] Quinlan A.R. Grabix. https://github.com/arq5x/grabix, 2014. [Online; accessed 21-January-2014].

[46] F. Chang et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, 2008.

[47] Hbase. http://hbase.apache.org, 2014. [Online; accessed 21-January-2014].

[48] C. Kozanitis et al. Using genome query language to uncover genetic variation. *Bioinformatics*, 20(1):1–8, 2014.

[49] J. Ostell et al. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. Wiley Interscience, second edition, 2001.

[50] O. Aparicio et al. Chromatin immunoprecipitation for determining the association of proteins with specific genomic sequences in vivo. *Current protocols in molecular biology / edited by Frederick M. Ausubel ... [et al.]*, Chapter 21, 2005.

[51] D.S. Johnson et al. Genome-wide mapping of in vivo protein-DNA interactions. *Science*, 316(5830):1095–9203, 2007.

[52] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 80–86, New York, NY, USA, 1983. ACM.

[53] M.T. Goodrich et al. External-memory computational geometry. In *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*, pages 714–723, Washington, DC, USA, 1993. IEEE Computer Society.

[54] H.P. Kriegel et al. The combination of spatial access methods and computational geometry in geographic database systems. In *Data Structures and Efficient Algorithms, Final Report on the DFG Special Joint Initiative*, pages 70–86, London, UK, 1992. Springer-Verlag.

[55] M. Suess and C. Leopold. A users experience with parallel sorting and OpenMP. In *Proceedings of Sixth European Workshop on OpenMP*, EWOMP '09, pages 23–28. Springer-Verlag, 2004.

[56] J.T. Robinson et al. Integrative genomics viewer. *Nature Biotechnology*, 29:24–26, 2011.

[57] A. McKenna et al. The genome analysis toolkit: a mapreduce framework for analyzing next-generation DNA sequencing data. *Genome Research*, 20(9):1297–1303, Sep 2010.