ABSTRACT

Title of Document:	ALGORITHMS AND HIGH PERFORMANCE COMPUTING APPROACHES FOR SEQUENCING-BASED COMPARATIVE GENOMICS
	Benjamin Thomas Langmead, Doctor of Philosophy, 2012
Directed By:	Professor Steven L. Salzberg Department of Computer Science

As cost and throughput of second-generation sequencers continue to improve, even modestly resourced research laboratories can now perform DNA sequencing experiments that generate hundreds of billions of nucleotides of data, enough to cover the human genome dozens of times over, in about a week for a few thousand dollars. Such data are now being generated rapidly by research groups across the world, and large-scale analyses of these data appear often in high-profile publications such as Nature, Science, and The New England Journal of Medicine. But with these advances comes a serious problem: growth in per-sequencer throughput (currently about 4x per year) is drastically outpacing growth in computer speed (about 2x every 2 years). As the throughput gap widens over time, sequence analysis software is becoming a performance bottleneck, and the costs associated with building and maintaining the needed computing resources is burdensome for research laboratories. This thesis proposes two methods and describes four open source software tools that help to address these issues using novel algorithms and high-performance computing techniques. The proposed approaches build primarily on two insights. First, that the Burrows-Wheeler Transform and the FM Index, previously used for data compression and exact string matching, can be extended to facilitate fast and memory-efficient alignment of DNA sequences to long reference genomes such as the human genome. Second, that these algorithmic advances can be combined with MapReduce and cloud computing to solve comparative genomics problems in a manner that is scalable, fault tolerant, and usable even by small research groups.

ALGORITHMS AND HIGH PERFORMANCE COMPUTING APPROACHES FOR SEQUENCING-BASED COMPARATIVE GENOMICS.

By

Benjamin Thomas Langmead

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2012

Dedication

To Sara.

Acknowledgements

Those friends thou hast, and their adoption tried,

Grapple them unto thy soul with hoops of steel.

(Hamlet 1.3.62-3)

I am greatly indebted to many friends, colleagues and mentors at the University of Maryland and Johns Hopkins University without whose support, friendship and understanding I would certainly not have finished this degree. My sincerest thanks go to Steven Salzberg, Mihai Pop, Rafael Irizarry, Michael Schatz, Cole Trapnell, Adam Phillippy, David Kelley, James White, Saket Navlakha, Jeffrey Leek, Kasper Hansen, Margaret Taub, Hector Corrada Bravo, and Andrew Feinberg.

I am also thankful to my previous employer, Reservoir Labs, Inc. Reservoir gave me my first taste of research, and of how satisfying it can be to work with brilliant colleagues. The trust and responsibility I was given there helped me grow into a much more effective researcher. My thanks go to all my colleagues at Reservoir, especially Richard Lethin.

Finally, I owe a great deal of thanks to my wife Sara and to my family: Mom, Dad, Missy and Greg. I love you all.

Table of Contents

Dedication ii
Acknowledgements iii
List of Tables vi
List of Figures vii
Chapter 1: Motivation 1
The technology 1
The computational problem 2
The growth problem 5
Improvements to read alignment 6
Comparative genomics approaches for "big data" 7
Chapter 2: Alignment using the Burrows-Wheeler Transform and FM Index 10
Local alignment and Smith-Waterman 10
Smith-Waterman is impractical 12
Indexing 13
The Burrows-Wheeler Transform 14
EXACTMATCH and the FM Index 17
Adding inexactness to EXACTMATCH 22
Excessive backtracking 25
Phased 2-mismatch search 27
MAQ-like search 29
Backtracking limit 31
Index construction 31
Index components 34
Performance results 34
Comparison to SOAP and MAQ 35
Read length and performance 38
Parallel performance 39
Bowtie software 40
Chapter 3: Extending Bowtie to finding longer, gapped alignments 42
Multiseeding: a better seed heuristic for longer reads 42
Support for gapped alignment 44
Gaps confound pruning 45
A new division of labor 47
Bowtie 2 workflow 48
Seed extraction 50
FM Index-assisted seed alignment 50
Seed alignment prioritization 51
SIMD-accelerated dynamic programming 52
Performance comparison on real data 55
Accuracy and sensitivity comparison on simulated data 61
Comparison of Bowtie 1 and Bowtie 2 64
Comparison to additional tools 68

Conclusions 69
Chapter 4: Crossbow, a "big data" approach to alignment and variant calling 71
A "big data" approach to comparative genomics 71
MapReduce 74
Steps in the Crossbow pipeline 74
Modifications to existing software 76
Cloud support 77
Performance results using simulated data 78
Results using whole-genome human resequencing data 80
Crossbow performance using cloud clusters 83
Conclusions 86
Chapter 5: Myrna: a "big data" approach to differential gene expression 87
RNA-seq 87
Variability in RNA-seq 88
Design of Myrna 89
Steps in the Myrna pipeline 91
Using MapReduce for permutation testing 94
Cloud computing performance 95
Conclusions 99
Bibliography 101

List of Tables

Table 1: Index-building memory footprint and wall clock time when indexing the whole human genome under various parameters. 33
Table 2: Performance results comparing Bowtie to SOAP and MAQ. 36
Table 3: Performance results comparing Bowtie to MAQ with filtered reads. 37
Table 4: Performance results comparing Bowtie to SOAP and MAQ across read lengths. 38
Table 5: Performance results when running Bowtie with various numbers of alignment threads. 39
Table 6: Speed and percent reads aligned for Bowtie 2 versus others: full results. 61
Table 7: Comparison of Bowtie 1 and Bowtie 2 on unpaired reads: full results. 66
Table 8: Comparison of Bowtie 1 and Bowtie 2 on paired-end reads: full results. 68
Table 9: Comparison of Bowtie 2 with various other tools. 69
Table 10: Experimental parameters for Crossbow experiments using simulated reads from human chromosomes 22 and X. 79
Table 11: Comparison of Crossbow and SOAP/SOAPsnp to Illumina 1M genotyping assay. 82
Table 12: Timing and cost for Crossbow experiments using reads from the Wang *et al.* study. 85
Table 13: Timing and cost for Myrna experiments using reads from the Pickrell *et al.* study. 98

List of Figures

Figure 1: Computing BWT(T) from T. 15
Figure 2: LF algorithm. 17
Figure 3: UNPERMUTE algorithm. 17
Figure 4: Illustration of UNPERMUTE. 17
Figure 5: LFC algorithm. 18
Figure 6: EXACTMATCH algorithm. 18
Figure 7: Illustration of EXACTMATCH. 19
Figure 8: Illustration of a slow algorithm for resolving a reference offset. 20
Figure 9: Illustration of a memory-intensive algorithm for resolving a reference offset. 21
Figure 10: A hybrid algorithm for resolving a reference offset. 22
Figure 11: Example of how exact and 1-mismatch algorithms might proceed. 25
Figure 12: Two cases considered by Bowtie searching for 1-mismatch alignments. . 27
Figure 13: Three cases considered by Bowtie searching for 2-mismatch alignments. 28
Figure 14: Example of how seeds are extracted in a multiseed scheme. 43
Figure 15: Examples of ungapped and gapped alignments, and gap types. 44
Figure 16: Gaps increase the number of ways we can proceed from a search node... 46
Figure 17: Steps of the Bowtie 2 workflow 49
Figure 18: Highlighted cells can be filled in parallel. 53
Figure 19: SIMD addition of two vectors of 8 16-bit values each. 54
Figure 20: Speed and percent reads aligned for Bowtie 2 versus others. 58
Figure 21: Alignment ROC curves for Bowtie 2 and other tools 63
Figure 22: Comparison of Bowtie 1 and Bowtie 2 on unpaired reads. 65
Figure 23: Comparison of Bowtie 1 and Bowtie 2 on paired-end reads. 67
Figure 24: Illustration of a “C to A” SNP 73
Figure 25: A sorted Crossbow bin ready for SNP calling 76
Figure 26: Illustration of how Crossbow is run using cloud resources. 78
Figure 27: Crossbow speedup versus number of processor cores allocated. 86
Figure 28: Illustration of the steps in the Myrna pipeline. 92
Figure 29: Myrna speedup versus number of processor cores allocated. 99

Chapter 1: Motivation

The technology

DNA sequencing technology has made huge strides since the earliest breakthroughs made by Frederick Sanger and others in the 1970s. Commercial entities including Life Technologies, 454 Life Sciences, and Illumina now manufacture and sell sequencers that, given a prepared biological sample, automatically report vast numbers of “reads”, i.e. snippets of DNA, present in the sample [1]. Other work shows that modern DNA sequencers can do much more than sequence genomes; they can be used to measure other cellular phenomena such as abundance of messenger RNA molecules [2], DNA methylation status [3, 4], and transcription factor binding [5].

These advances have spurred changes in the life sciences in recent years. In the 1990s, large-scale DNA sequencing was more or less the exclusive domain of large, well-resourced consortia like the Human Genome Project or companies like Celera Corporation. Now DNA sequencing, especially “second-generation” approaches such as sequencing-by-synthesis and sequencing-by-ligation [1], is extremely common and accessible even to small research laboratories. While sequencing instruments are costly (the Illumina instrument costs about $700,000), it is increasingly common for institutions to build “core facilities” where many sequencers and related instruments are maintained and run side-by-side and can be allocated to researchers on demand. The cost of the chemicals and other materials needed to run a modern sequencer is on the order of a few thousand dollars per run, and a HiSeq 2000 sequencer, for example, generates on the order of a few hundred billion DNA characters (henceforth called “nucleotides” or abbreviated “nt”) over the course of a single ten-day run [6]. It is informative to contrast these numbers with the prevailing conditions at the time of the commercial human genome project completed in 2000 by Celera Genomics, which reportedly cost around $300M, took about 2-3 years, and produced about 15 billion nucleotides of data [7]. The international Human Genome Project was yet more expensive and took more than a decade to complete [8].

The computational problem

Using a DNA sequencer is both like and unlike using a microscope to look at a cell. Like a microscope, a sequencer is a tool for investigating the contents of the cell. But whereas a microscope reveals visible parts of the cell, DNA sequencing addresses specific questions about parts of the cell that are not visible, e.g., what the sequence of its genome is, what messenger RNA molecules are present and in what abundance, or which parts of the DNA are bound by transcription factor proteins. These phenomena are critical to our understanding of how cells work, ultimately revealing much about health, disease, and evolution.

But another difference lies in how output is interpreted. When looking through a microscope, light photons bounce off the cell and travel through the microscope lens into the eye. Our brain receives the light signals and composes the image we see. Instead of producing photons, sequencers produce billions of “reads,” i.e. snippets of DNA that might be anywhere from a few dozen to several thousands of nucleotides (DNA characters) long. Each read is a copy of a snippet of DNA present in the cell.

Note that the sequencer does not simply report a completed “image,” i.e. an entire DNA molecule, such as a chromosome, from one end to the other. Though this would certainly be desirable, it is beyond the reach of current technology. Rather, to obtain a complete picture we must take a huge collection of relatively short sequencing reads and “glue” them together computationally.

To use another analogy, composing reads into a completed genome is like assembling a jigsaw puzzle. Both tasks involve identifying similarities: two puzzle pieces probably fit together if the images on the pieces match up; likewise, two reads probably came from overlapping places in the genome if the last several nucleotides in one read are similar to the first several nucleotides in the other read. Composing a set of reads into a genome by finding overlaps between reads and fitting them together is called “*de novo* assembly” or “assembly.” This task is related to the classic Shortest Common Superstring problem in computer science. Because of its usefulness for interpreting sequencing data, *de novo* assembly has become an important subfield of computer science and bioinformatics, and many approaches have been proposed and many software tools invented for doing this accurately and efficiently [9, 10].

While *de novo* assembly is an important tool, it is a very common practice to bypass assembly and instead utilize a template or “reference” genome as a shortcut. A reference genome is a previously assembled genome of the same species as the organism being sequenced. Using a reference genome as a guide is analogous to using the picture of the completed jigsaw puzzle printed on the box lid as a guide. The success of this approach hinges on a fortuitous property of genetics: two individuals of the same species tend to have extremely similar genome sequences. For instance, genomes of two unrelated humans are roughly 99.8 – 99.9% similar when measured as the fraction of aligned nucleotides that match when the two genomes are globally aligned [11].

This suggests a general approach to interpreting sequencing reads when a reference genome is available: for each read, determine the substring of the reference genome that is most similar to the read sequence. The place where this substring occurs constitutes our best guess as to where the read originated. The process of finding this reference substring is called "read alignment," or, depending on the context, "short read alignment," "read mapping," or simply "alignment" or "mapping." The general approach of using a reference genome to inform analysis is "comparative genomics." Thanks to the Human Genome Project and similar projects for other species (including model species like chimpanzee, mouse, rat, and yeast), many reference genomes are now available for use in comparative genomics projects. Comparative genomics is usually much less computationally expensive than *de novo* assembly. For this reason, comparative approaches are very popular, and *de novo* approaches are used chiefly to study species for which no reference genome yet exists.

de novo and comparative approaches share a key trait: both involve solving a computationally expensive set of sequence similarity problems. *de novo* assemblers find similarity between reads to determine how they overlap. This is computationally expensive because it involves considering similarities between all pairs of reads. Comparative genomics approaches find similarity between each read and the reference genome. This is also expensive because the reference is often very long; the human reference genome, for instance, is about 3 billion nucleotides long. In either case, naive approaches are far too slow.

While both the *de novo* and comparative problems are computationally challenging and worthy of attention, this thesis focuses exclusively on problems in comparative genomics. This is an important research trust because comparative approaches are very common in practice.

The growth problem

Comparative genomics involves solving many computationally expensive sequence similarity problems. We might take some comfort in the fact that computers get faster over time. Moore's Law is a rule of thumb stating (in one of its forms) that computer processor speed doubles about every eighteen months [12]. Thus Moore's Law might also be said to describe the rate at which "computational throughput" increases over time[1].

Sequencers, on the other hand, are getting faster at a rate that far outpaces Moore's Law. For instance, a series of press releases describing the Illumina Genome Analyzer sequencer and its successor, the HiSeq 2000, show a trend whereby per-sequencer throughput grew at a rate close to 4x per year over 2009 and 2010 [13, 14]. A similar trend can be observed for the Life Technologies SOLiD sequencer. In addition, sequencing costs are rapidly decreasing [15], and the overall number of sequencers sold is increasing [16].

[1] There are other aspects of computer architecture besides processor speed that can impact analysis speed: input/output speed for example. In practice, comparative genomics is quite processor-intensive, so it is reasonable to use Moore's Law as a stand-in for computational throughput here.

With these trends comes an urgent need for better computer science methods. More efficient software algorithms and high-performance implementations of those algorithms must be invented and made available to life science researchers. If such improvements are not forthcoming, sequencing-driven research pipelines will be forced to stall while slower analyses catch up with faster sequencers. This is the chief motivation behind the methods and software tools described in this thesis.

Improvements to read alignment

This thesis addresses the large and growing gap between sequencing throughput and computational throughput in two complementary ways. First, I propose a novel indexing strategy and novel alignment algorithms that enable very fast alignment of DNA sequencing reads to large genomes. The specific novel contributions in this thesis include: (a) the idea of using the FM Index [17] as a time- and space-efficient full-text index to aid biological sequence alignment, (b) the idea that inexact matching with the aid of a full-text index such as the FM Index can be framed as a search over the space of possible ways of mutating the query (read) string into a string that occurs in the text (reference), (c) a set of techniques for pruning the search space so that it can be explored efficiently, including reference pruning, policy pruning, and double indexing, and (d) the idea of supporting fast gapped read alignment by separating alignment into an initial full-text-assisted seed alignment phase followed by a hardware-accelerated dynamic-programming-based extension phase.

Items (a), (b) and (c) were first implemented in Bowtie [18], an open source software tool for finding ungapped alignments of short DNA sequencing reads (about

25-50 nt long) to large genomes. Bowtie was released to the public in 2009. At that time, Bowtie outperformed competing approaches in terms both of speed and of memory efficiency. The methods underlying Bowtie will be discussed in Chapter 2, but additional details regarding the design of Bowtie, as well as performance results and comparisons to other tools, are presented in published papers including the original Bowtie paper [18], the protocol paper [19], and in surveys [20].

Items (a), (b) and (c) and (d) were implemented together in Bowtie 2. Bowtie 2 is an open source software tool for finding gapped alignments for DNA sequencing reads up to about 10,000 characters long. Bowtie 2 addresses the most pressing shortcomings of Bowtie: its inability to find alignments with gaps and its poor performance when aligning longer reads. Bowtie 2 does this efficiently using a two-phase approach whereby the FM Index is used to find ungapped seed alignments, then a hardware-accelerated dynamic programming algorithm is used to find full, gapped alignments. Bowtie 2 was released to the public in 2011 and the methods underlying Bowtie 2 are discussed in Chapter 3.

Comparative genomics approaches for "big data"

While Bowtie and Bowtie 2 represent an important step toward closing the gap between sequencer throughput and analysis throughput, we cannot in general depend on algorithmic improvements to close this gap now and in the future. Another option for increasing throughput is to run analyses on many processors at once.

Parallel and high-performance computing approaches, both simple parallel approaches (e.g. straightforward use of multicore servers or Sun Grid Engine) and more elaborate parallel approaches [21-23] have been applied to genomics in the past.

However, when one considers the sheer size of the sequencing datasets generated today, and the rate at which sequencing throughput is likely to grow over time, parallelization is just one issue among many that come to the fore. For instance, consider an analysis of a very large sequencing dataset that takes several weeks to run on many hundreds of computers in parallel. A severe hardware or software failure on any one of those computers could cause the entire computation to fail, potentially invalidating hundreds or thousands of computer-hours of work.

The terms “big data” and “data intensive computing” describe the set of issues that become crucial when datasets and computations become very large [24]. Key “big data” issues include (a) parallelization: the ability to divide a computation up into many small, independent pieces that can be executed simultaneously on different processors, (b) fault tolerance: the ability to recover from unexpected and severe failures, and (c) economy: the need to keep subtasks small so that they can execute on a variety of different types of computer without exceeding available resources or crowding out other subtasks. In the past, solutions to these problems have been put together in an *ad hoc* fashion, with pieces provided by software developers, users, and their system administrators. But as datasets continue to grow, and as the user community continues to widen to include more users with limited computational resources and savvy, it is increasingly critical for “big data” features to be built directly into everyday software tools.

In Chapters 4 and 5 of this thesis, I describe two open source software tools that I developed in collaboration with colleagues at University of Maryland and Johns Hopkins University. These tools, Crossbow and Myrna, solve common comparative genomics problems while also addressing “big data” issues. Crossbow and Myrna are notable primarily because they demonstrate how common comparative genomics tasks can be adapted to run in the restrictive MapReduce programming framework. The framework in turn provides a number of services, including redundant, distributed data storage and flexible, parallel execution of subtasks.

Crossbow and Myrna are also notable because they can be run using computers rented from the Amazon Web Services [25] cloud computing service. Crossbow and Myrna tools were among the first examples of useful software tools (the first being CloudBurst [26]) designed to exploit the benefits of cloud computing – a model whereby computing resources and services are rented from large providers. In the future, cloud computing could be a useful paradigm for life science labs struggling to keep pace with sequencers because: (a) cloud computing allows users to rent large collections of computing resources over the Internet, potentially freeing researchers from the burdens of building and maintaining their own computing resources, (b) the MapReduce parallel programming model [27], which is well suited to run on clusters of computers rented from a cloud vendor, is also a good fit for many comparative genomics applications, and (c) cloud computing’s use of virtualized hardware and software makes it easier for researchers to agree on standards that promote usability and reproducibility of analysis software. See published surveys for more detailed arguments for and against cloud computing [15, 28, 29].

Chapter 2: Alignment using the Burrows-Wheeler Transform and FM Index

This Chapter introduces the read alignment problem, illustrates why a classic Smith-Waterman-style dynamic programming approach is too inefficient for this application, introduces indexing as a way of accelerating alignment, then introduces the Burrows-Wheeler Transform, the FM Index, and other indexing and search methods underlying Bowtie. I designed Bowtie, and the Bowtie software implementation is also primarily my work though Cole Trapnell helped with software implementation. Bowtie was originally published in 2009 [18], and a protocol paper was published in 2010 [19].

Local alignment and Smith-Waterman

The chief computational problem underlying comparative genomics is read alignment: given a sequencing read, we would like to find the substring of the reference sequence that is most similar to the read. The location of this best match is our best theory for where the read originated with respect to the reference sequence. This problem is closely related to the classic local alignment problem:

> Given a string P ("pattern") of length m and a string T ("text") of length n, find substrings a and b of P and T respectively having maximal optimal global alignment score.

"Global alignment score" is a similar to edit distance. The edit distance between two strings is the minimal number substitutions and gaps that must be introduced to transform one string into the other. A global alignment score additionally associates scores with matches, substitutions and gaps. Consider this example (from Gusfield [30], p. 230) where matches have a score of +2 each, mismatches a score of -2 each and gaps a score of -1 each:

P = `xyaxbacsll`, T = `pqraxabcstvq`

The optimal local alignment is:

```
a: a x — b a c s
b: a x a b — c s
```

Where dash (“-“) represents a gap. The global alignment score of this local alignment is 8, since there are 5 matches (contributing +10), two gaps (contributing -2), and no mismatches.

A well studied algorithm for finding an optimal solution to the local alignment problem is Smith-Waterman [31], or the Smith-Waterman-Gotoh [32] variant thereof. Smith-Waterman proceeds by filling in a dynamic programming matrix where rows correspond to positions in P and columns correspond to positions in T. Each cell value is set to the global alignment score of the best local alignment of the prefixes of P and T corresponding to (i.e. up to and including) the cell’s row and column.

This problem has optimal substructure; a given cell’s value depends only on the values of its neighboring cells above, to the left, and to the upper-left in the matrix. Once the matrix is filled in completely, the best alignment is obtained via a backtrack procedure. The time complexity is O(mn), and the space complexity is also O(mn), though variants exist that achieve better space bounds. For more details, including a proof of optimality, see Gusfield [30], p. 230.

Smith-Waterman is impractical

Consider the performance of Smith-Waterman for the case where the input consists of 6 billion sequencing reads of length m = 100 and the reference sequence is the human reference genome, i.e. n = ~ 3 billion nucleotides. This situation corresponds to a typical run of the Illumina HiSeq 2000 sequencer. The total number of cell updates required to fill all dynamic programming tables for all reads is 6 billion (reads) times 100 (nucleotides per read) times 3 billion (reference sequence nucleotides) = 18 x 10^{20} or about 2 x 10^{21}. Say that we have 1,000 processors, each clocked at 6 gigahertz and capable of completing a single Smith-Waterman cell update every clock cycle. Such a collection of processors has an aggregate throughput of 6 x 10^{12} Smith-Waterman CUPS (Cell Updates Per Second) and requires about 3 x 10^{8} (18 x 10^{20} divided by 6 x 10^{12}), seconds or about 9 years to complete the computation.

This demonstrates that Smith-Waterman is not fast enough for second-generation sequencing workloads. Improvements in efficiency over Smith-Waterman are mostly achieved by reducing the impact of either the m term or the n term in Smith-Waterman's O(mn) time complexity. MAQ [33], for example, reduces the impact of the m term by bundling reads into sets, creating a hash table-based index over the set, then traversing the reference sequence only a handful of times for each set. Bowtie [18], BWA [34], BWA-SW [35], SOAP [36], SOAP2 [37], and many other modern tools seek to reduce the impact of the n term using a pre-computed index of the reference genome. A common theme is the use of indexing to achieve better performance.

Indexing

An index summarizes a text in a way that (a) can be queried rapidly, and (b) allows us to narrow our focus to a small number of interesting candidate locations. This is analogous to the index of a book. A book index is a list of key terms extracted from the text of the book, where alongside each key term is a list of page numbers where the term is used. The index is sorted in alphabetical order to allow the reader to look up terms efficiently.

While some tools choose to index batches of reads and leave the reference as-is, tools that index the reference genome have often proven to be faster. While building the index takes a significant amount of time (6-8 hours to build an index for the human reference genome using Bowtie), this cost is usually ignored when discussing the performance of the algorithm because it is incurred once per reference genome. That cost is then amortized over every alignment job that uses that index.

Many indexing and index querying schemes for alignment have been proposed, including schemes that employ keyword tables (also called "seed tables"), spaced-seed tables [33, 36, 38], q-gram filtering [39-41], suffix trees [42, 43] and suffix arrays [43-45]. When used to index reference genomes as long as 3 billion nucleotides, though, these approaches incur large memory footprints. A suffix array, for example, must store one integer per reference character, with each integer occupying O(log(n)) bits. For the human genome, such an index occupies about 3 billion (characters) times 4 bytes (one 32-bit offset per character) = 12 gigabytes. Suffix trees are typically even larger, on the order of a few dozen bytes per reference character, or about 20 bytes per character if optimized for space [46]. Tables such as spaced-seed tables could be small or large, but are often more than 10 gigabytes for the human genome in practice (e.g. about 13 gigabytes for SOAP [36]). One of the original goals for Bowtie was to fit within the memory budget of a typical desktop computer.

The Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) of a text is a reversible permutation of its characters. Originally developed for data compression [47], BWT-based indexing allows large texts to be searched efficiently and in a small memory footprint. Prior to Bowtie's publication, the BWT and the related FM Index [17] had been applied previously to bioinformatics applications including oligomer counting (i.e. counting the number of times substrings of a certain length occur in a sequence) [48], whole-genome alignment [49], tiling microarray probe design [50], and Smith-Waterman alignment to a large reference [51]. Since the publication of Bowtie, other read alignment tools based on the BWT have also been published, including BWA [34], BWA-SW [35] and SOAP2 [37].

The Burrows-Wheeler Transform of a text T, BWT(T), can be constructed as follows. The character \$ is appended to T, where \$ is a character not in T that is lexicographically less than all characters in T. The Burrows-Wheeler Matrix of T, BWM(T), is obtained by computing the matrix whose rows comprise all cyclic rotations of T sorted lexicographically. BWT(T) is the sequence of characters in the rightmost column of BWM(T) (Figure 1).

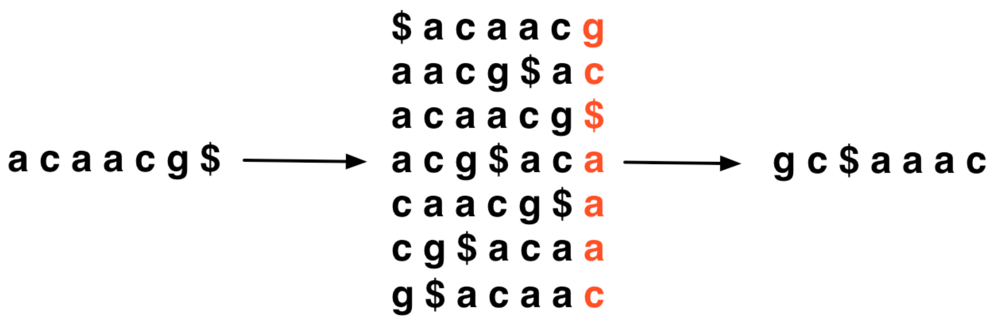

Figure 1: Computing BWT(T) from T.

A Burrows-Wheeler matrix has a property called the Last First ("LF") Mapping. The property is: the ith occurrence of character c in the last column of the matrix corresponds to the same text character as the ith occurrence of c in the first column. For instance, in Figure 1, the second A in the last column and the second A in the first column both correspond to the first A in T. Burrows and Wheeler prove the property as follows. Given a Burrows-Wheeler Matrix M, construct matrix M' by cyclically rotating all rows of M to the right by one position. By construction, M' is the matrix of all cyclic rotations of T sorted lexicographically and cyclically starting at their second (not first) character. Consider just the rows of M' beginning with character c. These rows must appear in lexicographical order with respect to each other; they are "tied" with respect to their first character and sorted with respect to their second. For a character c, rows beginning with c in M appear in the same order as rows beginning with c in M'. Since the first column of M' is the same as the last column of M, the LF Mapping property follows.

The LF mapping underlies key algorithms that use BWT(T) to navigate or search in T. The UNPERMUTE algorithm applies it repeatedly to re-create T from BWT(T). Consider a function LF(r) that, given row index r into the Burrows-Wheeler Matrix, returns the index of the corresponding row according to the LF mapping property. For instance, if the last character of row r is the j^{th} occurrence of character c in the last column, LF(r) returns the index of the row containing the j^{th} occurrence of c in the first column. Since the first character of row LF(r) corresponds to the same text character as the last character of row r, the last character of row LF(r) must correspond to the text character that cyclically precedes that character in the text. By applying r = LF(r) repeatedly starting in the row whose last character corresponds to the last character of T (i.e. the row beginning with $), we can follow the sequence of Burrows-Wheeler rows corresponding to consecutive text characters from right to left. We recreate T by performing this walk and aggregating the visited text characters in a buffer.

LF(r) can be implemented in terms of an array C[c] and a function Occ(c, r) as shown in Figure 2. Elements of C are pre-calculated so that C[c] equals the total number of occurrences of all alphabet characters lexicographically less than character c in T. The Occ(c, r) function counts the number of occurrences of character c in a prefix of BWT(T) up to but not including the r^{th} character. UNPERMUTE is implemented in terms of LF(r) as shown in Figure 3 below. Figure 4 illustrates how the UNPERMUTE algorithm reconstructs the original string ACAACG$ from the permuted string GC$AAAC.

```
LF($r$):
    c ⇐ $BWT[r]$
    return $C[c]$ + Occ($c$, $r$) + 1
```

Figure 2: LF algorithm.

```
UNPERMUTE:
    $r$ ⇐ 1
    $T$ ⇐ empty string
    while $BWT$[r] ≠ $ do
        prepend $BWT[r]$ to $T$
        $r$ ⇐ LF($r$)
    end while
    return $T$
```

Figure 3: UNPERMUTE algorithm.

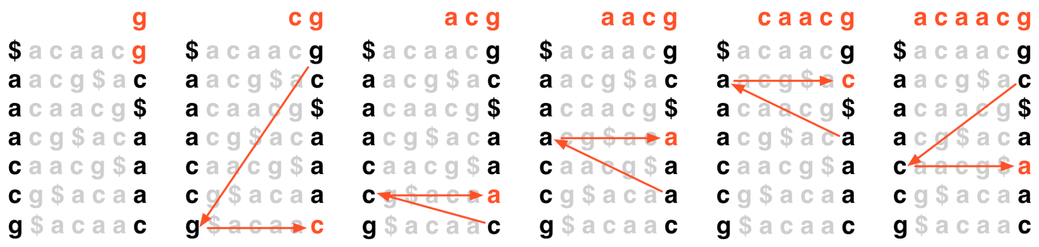

Figure 4: Illustration of UNPERMUTE.

EXACTMATCH and the FM Index

Ferragina and Manzini observe that the Burrows-Wheeler Transform and the LF Mapping can also be used to perform exact matching of a query string P within the text T [17]. Because the rows of the Burrows-Wheeler Matrix are sorted lexicographically, all rows having P as a prefix must be consecutive. The EXACTMATCH algorithm (Figure 6) iteratively calculates ranges of Burrows-Wheeler rows prefixed by successively longer suffixes of the query. At each step, the length of the suffix under consideration grows by one character and the size of the range either shrinks or remains the same. Like UNPERMUTE, EXACTMATCH makes use of a helper function based on the LF mapping, called LFC. Unlike LF, LFC takes a second argument c, where c is a character drawn from the text alphabet. LFC performs the same calculation as LF, but as though the character in the last column of row r is c, which it may or may not be. LFC is shown in Figure 5. Figure 7 illustrates of the steps taken by the EXACTMATCH algorithm to match the pattern AAC in the text ACAACG. The correctness of EXACTMATCH is established in appendix B of Ferragina and Manzini's paper [17].

```
LFC($r$, $c$):
    return $C[c]$ + Occ($c$, $r$) + 1
```

Figure 5: LFC algorithm.

```
EXACTMATCH($P[1, p]$):
    $c \Leftarrow P[p]$
    $sp \Leftarrow C[c] + 1$
    $ep \Leftarrow C[c + 1] + 1$
    $i \Leftarrow p - 1$
    **while** $sp < ep$ and $i \geq 1$ **do**
        $c \Leftarrow P[i]$
        $sp \Leftarrow$ LFC($c$, $sp$)
        $ep \Leftarrow$ LFC($c$, $ep$)
        $i \Leftarrow i - 1$
    **end while**
    return $sp$, $ep$
```

Figure 6: EXACTMATCH algorithm.

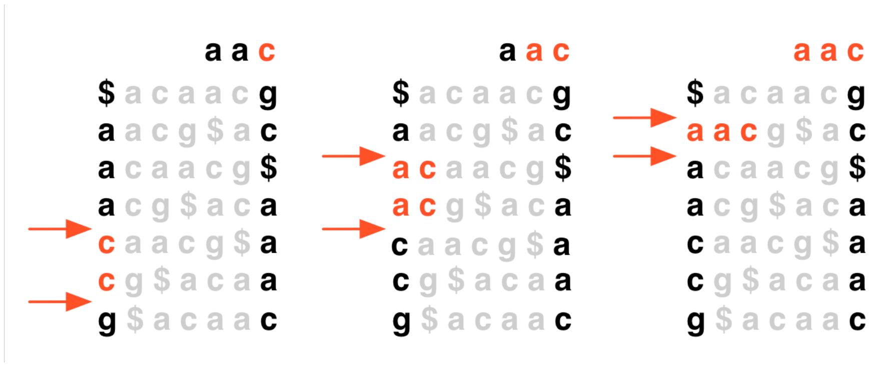
Figure 7: Illustration of EXACTMATCH.

We have yet to establish whether UNPERMUTE and EXACTMATCH scale well to large texts. A problem is that each call to LF(r) or LFC(r, c) triggers a call to Occ(c, r), which, naively implemented, examines a number of characters proportional to the length of T in the worst case. Ferragina and Manzini [17] propose accelerating Occ(c, r) by pre-calculating and storing character occurrence counts for each character in the alphabet up to certain regular positions throughout BWT(T). If the pre-calculated positions ("checkpoints") are chosen such that the space between consecutive checkpoints is bounded by a constant B, then an efficient implementation of Occ(c, r) need examine at most B characters of BWT(T) per call. Thus, Occ(c, r) can be made to operate in constant time at the cost of having to pre-calculate and store checkpoints that occupy space proportional to the length of T times the cardinality of the alphabet. Note that if Occ(c, r) is constant-time, the overall EXACTMATCH algorithm is linear-time in the length of the query P.

The final output of the EXACTMATCH algorithm is a range of matrix rows beginning with a given query string P, i.e., the rows delimited by the sp and ep variables from Figure 6. Each row corresponds to an exact match of P somewhere in the text, but more work is required to determine, for a given row, which text offset it corresponds to. One naive solution is: given row r, calculate r = LF(r) repeatedly zero or more times until r equals the row with $ in the last column. The original row's offset into the reference text equals the number of times LF(r) was called before reaching that row. A simple example is shown in Figure 8. This approach is not time-efficient, since calculating a row's offset requires a number of calls to LF that is linear in the length of T.

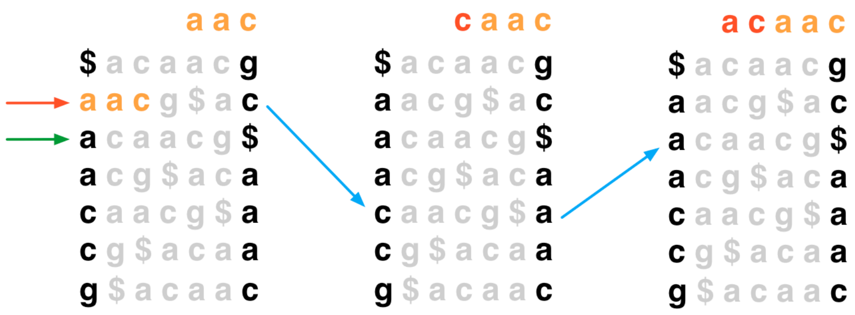

Figure 8: Illustration of a slow algorithm for resolving a reference offset.

Another naive solution is to, at index building time, pre-compute and store an array parallel to BWT(T) containing the reference offsets of each row. This array is simply the suffix array of T. To resolve the reference offset of row r, we look up element r in the pre-calculated array (see Figure 9). This solution is not space-efficient: if n is the length of T, storing the suffix array of T requires an amount of space proportional to O(n log(n)), which, for the 3-billion-nucleotide human genome reference sequence, requires about 12 gigabytes of storage.

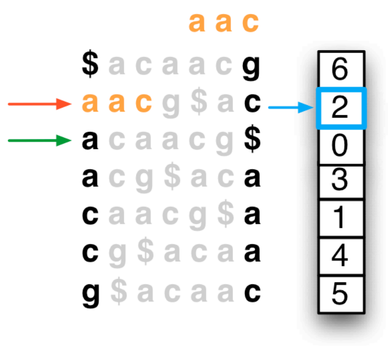

Figure 9: Illustration of a memory-intensive algorithm for resolving a reference offset.

Ferragina and Manzini [17] propose a hybrid scheme whereby a subset of the rows of the matrix are associated with pre-calculated text offsets. To retrieve a row r's text offset, we first check if r is one of the rows with a pre-calculated offset. If so, we retrieve and report the offset. If not, we calculate r = LF(r) repeatedly until r does correspond to a row for which the offset was pre-calculated, at which point we retrieve and report the pre-calculated offset for r plus the number of times LF(r) was called before reaching r. Pre-calculating offsets for a larger fraction of rows allows text offsets to be calculated faster on average, but pre-calculating a smaller fraction reduces the overall size of the index. Bowtie adopts this scheme with a default (but configurable) policy of pre-calculating offsets for every 32nd row. Figure 10 illustrates an example where LF(r) is called exactly once, causing the walk to enter a row with pre-calculated offset 1.

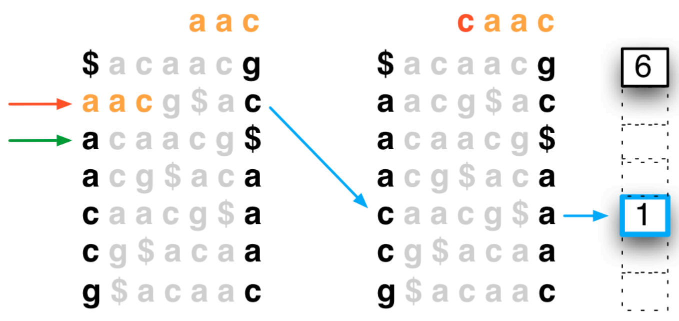

Figure 10: A hybrid algorithm for resolving a reference offset.

Bowtie’s scheme of storing pre-calculated offsets every 32 rows does not provide a worst-case guarantee better than O(n) for the number of times LF must be called to calculate the offset for a given row. Ferragina and Manzini’s originally proposed scheme does provide such a guarantee by selecting pre-calculated rows according to a regular periodic sample of characters in T (as opposed to Bowtie’s scheme of regularly sampling characters in BWT(T)). Bowtie’s scheme was selected for its simplicity and because the average number of calls to LF (as opposed to the worst-case number) should still be comparable to Ferragina and Manzini’s scheme.

It is also notable that, in a follow-up to their initial FM Index paper, Ferragina and Manzini propose a scheme that, like Bowtie’s, does not guarantee sublinear worst-case performance [52].

Adding inexactness to EXACTMATCH

EXACTMATCH itself is not sufficient for aligning sequencing reads because the best alignment for a read may contain mismatches and gaps. Differences may be due to sequencing errors, actual genetic differences between reference and subject organisms, or a combination of the two. To allow for differences, we design an algorithm that conducts a search through the space of possible alignments to quickly find those satisfying the desired alignment policy. Though a completely unpruned search space has size exponential in the length of the read, much pruning is possible in practice. My experiments indicate that this method is generally tractable for up to (at least) three mismatches in practice, yielding a very fast alignment algorithm for those cases. Though the technique can be made to deal with gaps as well as mismatches, the Bowtie implementation currently deals only with mismatches. Chapter 3 includes a detailed discussion of how support for gapped alignments can be added.

The search process makes use of numeric quality values on the Phred scale [53], where, if the sequencer's software predicts that the probability of a nucleotide having been miscalled is p, the Phred quality value of the nucleotide is reported as -10 log10 p. Quality values are used to assess the likelihood of candidate alignments under a model where all differences are assumed to be due to sequencing error. Quality values direct the search to the positions that are most likely to be wrong first.

Inexact search proceeds similarly to EXACTMATCH, calculating Burrows-Wheeler ranges for successively longer query suffixes. If the search arrives at an empty Burrows-Wheeler range, this indicates the suffix does not occur in the text. In this case, the algorithm may select a previously examined query position and substitute a different nucleotide there. This introduces a hypothetical mismatch into the alignment, and we call this a "backtrack." After executing a backtrack, the EXACTMATCH search resumes from just to the left of the substituted position.

The search performs only those backtracks that are consistent with the user-configurable alignment policy. For example, if the alignment policy imposes a maximum of two mismatches in the entire alignment and the search procedure is at a position P in the search space where two mismatches have already been hypothesized along the path from the root to P, and an empty Burrows-Wheeler range is obtained at P, the search will not attempt to hypothesize a third mismatch as doing so would violate the alignment policy. This is called “policy pruning.” Also, the search will never backtrack to points in the search space that are already known to be associated with empty Burrows-Wheeler ranges. Doing so would be fruitless, since there is no substring of the reference possessing the appropriate characters. Pruning partial alignments in this way is called “reference pruning.” Figure 11 illustrates a simple example of how exact and 1-mismatch search strategies might proceed for a read with a 1-mismatch alignment to the reference. In this case, the query GGTA does not have an exact match in the text, but does have a 1-mismatch alignment where a G in the reference is substituted for A in the read. Pairs of numbers represent the sp, ep pairs calculated in an iteration of EXACTMATCH. Vertical sets of four pairs represent the pairs calculated for A, C, G and T (top to bottom). Blue numbers and letters represent hypothetical mismatches introduced as part of a backtrack. Empty ranges are shown in red if they trigger a backtrack, or in gray if they do not. Empty boxes correspond to ranges left uncalculated due to policy pruning. The final reported range is shown in green. In practice, maximizing the amount of pruning possible is critical to minimizing the running time of the search.

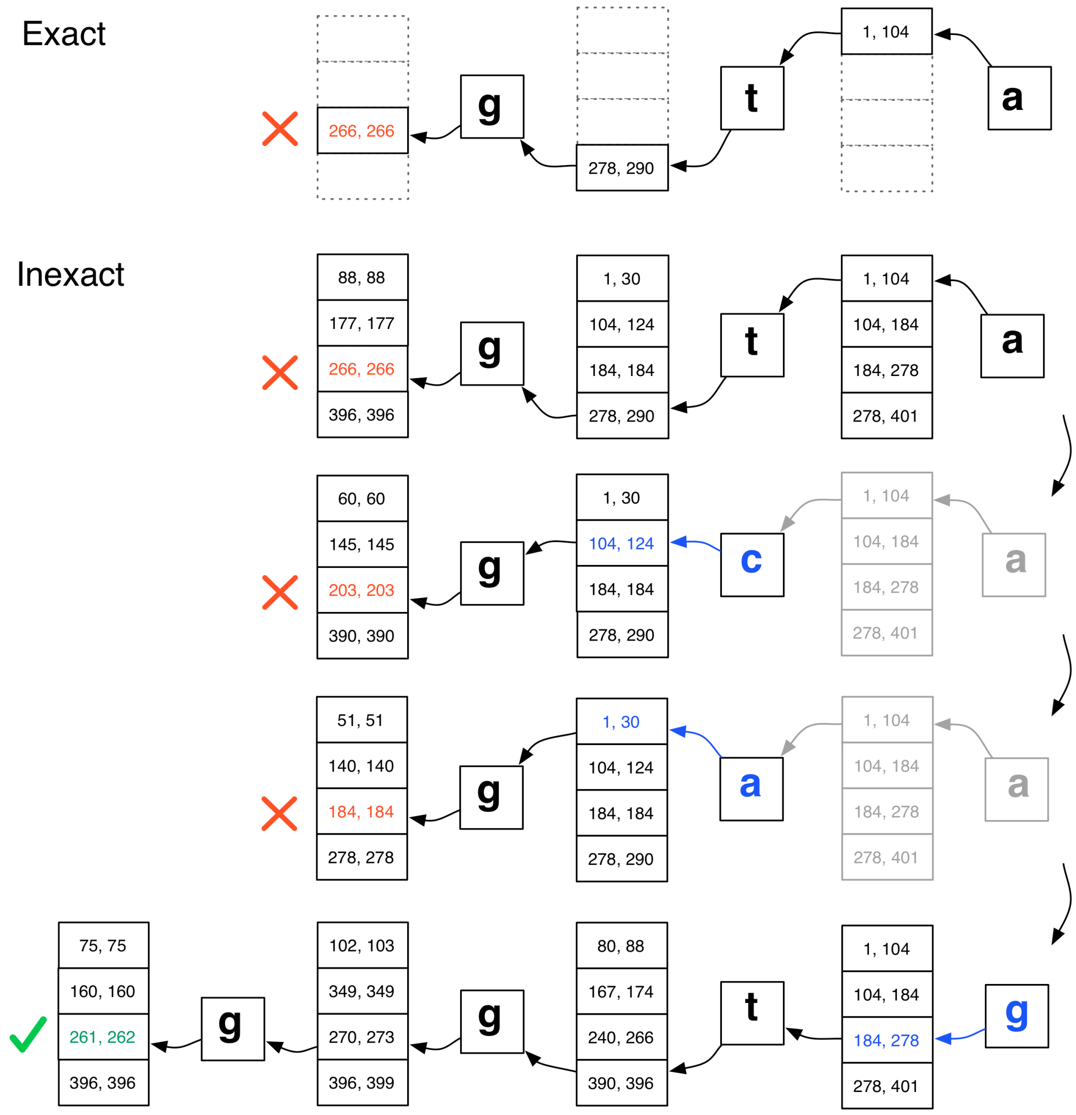

Figure 11: Example of how exact and 1-mismatch algorithms might proceed.

Excessive backtracking

The strategy described has the drawback that some inputs cause excessive backtracking. Excessive backtracking occurs when the two pruning strategies are not sufficient to prevent the aligner from performing enough backtracks to noticeably affect performance. Since short suffixes of the read (corresponding to the neighborhood around the root of the search space) are likely to occur in the reference simply by chance, excessive backtracking is particularly prevalent in first several levels (“ply”) of the search space. Consider an attempt to find an alignment with up to 2 mismatches for a 20-mer against a reference sequence containing every possible 10-mer (e.g. the human genome [54]). If no such alignment exists, the search is forced to explore all combinations of 2 backtracks within that first ten ply of the search space.

Bowtie mitigates excessive backtracking using “double indexing.” With double indexing, two Burrows-Wheeler indexes of the genome are created: one for the normal genome sequence, called the “forward index,” and a second for the reverse of that sequence (not the reverse complement), called the “mirror index.” To see how this helps, consider a matching policy that allows up to one mismatch in the alignment. A valid alignment falls into one of two cases according to which half of the alignment contains the mismatch; by convention, we lump the case where the alignment has no mismatches in with the first enumerated case. To identify alignments falling into case 1, where either there are no mismatches or the left half contains exactly one mismatch, we use the forward index and invoke the search routine with the constraint that it may not backtrack to any of the positions in the right half of the alignment. To identify alignments falling into case 2, where the right-hand side of the alignment contains exactly one mismatch, we use the mirror index and invoke the search routine on the read with its character sequence reversed, with the constraint that the aligner may not backtrack to positions in the right half of the alignment. Note that because the read sequence has been reversed, the right half of the alignment in case 2 corresponds to the left half in case 1. Figure 12 illustrates the two cases. Numbers shown below the alignment segments indicate permitted numbers of substitutions in those segments. By forbidding backtracks to positions close to the right-hand side (i.e. the root) of the alignment, this strategy avoids a great deal of backtracking.

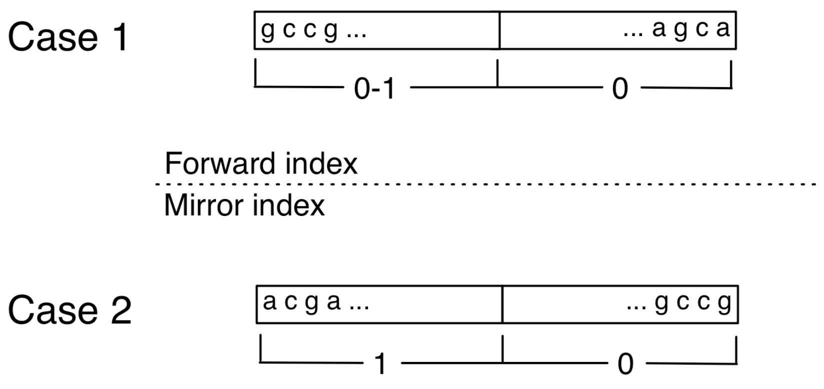

Figure 12: Two cases considered by Bowtie searching for 1-mismatch alignments.

Some excessive backtracking may still occur in the left half of the alignment, especially in those positions just to the left of the halfway mark. Still, results indicate double indexing performs well in practice.

<u>*Phased 2-mismatch search*</u>

Excessive backtracking becomes more problematic when the alignment policy permits 2 or more substitutions. This is because (a) even with double indexing, it is not possible to avoid allowing substitutions in the right half of the alignment in some cases, and (b) if two or more stretches of the alignment are permitted to contain a substitution and many substitutions are possible along two or more of those stretches, the number of potential backtracks is related to their product in the worst case. This multiplicative effect can drastically reduce performance.

Bowtie’s 2-mismatch search strategy is divided into three cases. Case 1 uses the forward index and constrains the right half of the alignment to contain no mismatches while the left half may contain up to 2 mismatches. Case 2 uses the mirror index (and the reversed read) and constrains the right half of the alignment to contain no mismatches while the left half may have either 1 or 2 mismatches. Case 3 uses the forward index and constrains the right and left halves to contain exactly one mismatch each. Figure 13 illustrates these cases. A 2-mismatch alignment can be uniquely assigned to one of these three cases.

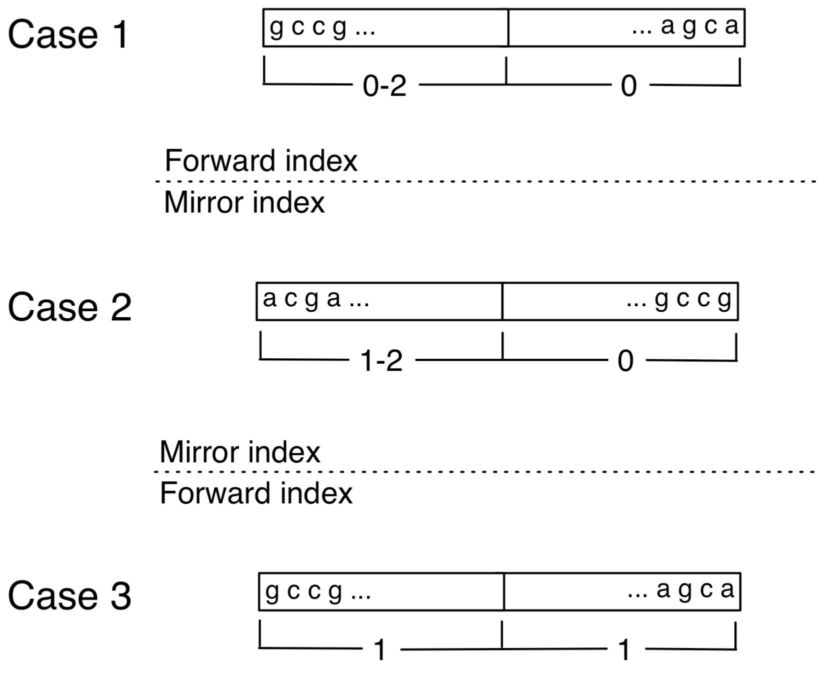

Figure 13: Three cases considered by Bowtie searching for 2-mismatch alignments.

The 3-case approach of Figure 13 allows substantial pruning. However, case 3 allows a mismatch in the right half of the alignment, and so is particularly vulnerable to excessive backtracking. In practice the overhead of excessive backtracking is onerous for some reads, but the overall running time of the search across many reads is good in practice.

The 3-mismatch case is an extension of the 2-mismatch case; discussion of the 3-mismatch case will be omitted here.

MAQ-like search

The search strategies described so far handle alignment policies where a few mismatches are permitted in the entire alignment and quality values are not considered. The family of alignment policies enforced by MAQ [33] is different in two ways. First, MAQ enforces a ceiling on the sum of the Phred [53] quality values at all mismatched positions. The default ceiling is 70. For example, an alignment with two mismatches, both at positions with Phred quality 30, is permissible by default. A similar alignment where both mismatched positions have Phred quality 40 is not permissible by default, since the sum, 80, exceeds the default ceiling of 70. Second, MAQ constrains the number of mismatches permitted, but only in the first several nucleotides of the read (the "seed"), not in the entire alignment. MAQ permits up to two mismatches in the first 28 nucleotides of the read by default. Any number of mismatches is permitted outside the seed, though the legality of the overall alignment is still subject to the quality ceiling.

These differences require changes to Bowtie's search strategy. First, Bowtie must keep track of the sum of the Phred quality values at positions where hypothetical mismatches have been introduced so far. Policy pruning must be broadened to additionally prune paths that, if followed, would violate the quality ceiling. Bowtie must also treat the seed and non-seed portions of the alignment appropriately. Since the seed is more constrained, an efficient strategy is for Bowtie to align the seed portion first, then relax the mismatch ceiling and extend the alignment through the non-seed portion. In essence, this is a "seed and extend" strategy.

The search strategies presented in previous sections, appropriately extended to handle the quality ceiling, suffice for aligning the seed portion. A complication arises when the index used to align the seed cannot be used to extend the seed. For example, if the seed portion of the read aligns to a single location on the reference and that alignment contains two mismatches in the left half of the seed, the 2-mismatch search strategy described previously will use the forward index to find the seed alignment (case 1), yielding a range of Burrows-Wheeler rows in the forward index. The non-seed portion of the read is to the right of the seed, but EXACTMATCH can only be used to extend right-to-left. To translate a range of rows in the forward index into a corresponding range of rows in the mirror index (or vice versa), Bowtie simply re-matches the string of characters that led to the initial range. Consider a situation where the read is the 9-character string TAACCCAGG, the seed length is 6, and aligning just the seed TAACCC yields a 1-mismatch alignment in the forward index where A in the reference is substituted for T in the seed. The range obtained by aligning the seed cannot be extended through the non-seed portion because the non-seed portion lies to the right in the context of the forward index. Bowtie handles this by switching to the mirror index and re-matching the string CCCAAA, which is the mirror image of the 6 seed characters including the substitution introduced during the seed alignment. Bowtie then relaxes the substitution limit and extends the alignment through the non-seed portion of the read in the usual way. Note that this scheme could be further improved by harnessing the "Bi-directional BWT" approach proposed by Lam *et al.* [55].

Backtracking limit

Even with the above measures, excessive backtracking can have a significant adverse impact on performance when a read has many low-quality positions and does not align or aligns poorly to the reference. These cases can trigger many hundreds of backtracks per read. This cost is mitigated in Bowtie by enforcing a limit on the number of backtracks allowed before search is terminated (default: 125 in the depth-first mode, 800 in best-first mode). The limit prevents some legitimate, low-quality alignments from being reported, but this tradeoff is desirable for most applications. The limit is only in effect when the alignment policy selected by the user is either the 2-seed-mismatch MAQ-like or 3-seed-mismatch MAQ-like policy.

Index construction

The central problem of building a Bowtie index lies in calculating the Burrows Wheeler Transform of the reference genome sequence. This is closely related to the problem of building a suffix array. Each element of the BWT is derivable from the corresponding element of the suffix array according to a simple formula:

$$BWT[i] = \begin{cases} T[SA[i] - 1] & SA[i] = 0 \\ \$ & SA[i] = 0 \end{cases} \tag{2.1}$$

A conceptually simple way of calculating BWT is to build a suffix array using a suffix-sorting technique (e.g. the approach of Manber and Myers [56] or Larsson and Sadakane [57]), then calculate the BWT through repeated applications of Formula 2.1 in a single pass over the suffix array. However, this incurs a very large memory footprint. Constructing the suffix array in memory uses at least about 12 gigabytes for the human genome, for example. This may be acceptable for users with access to large-memory computers, but our goal is to facilitate research on typical workstations, so a more memory-efficient solution is desirable.

Kärkkäinen [58] proposes a memory-conscious blockwise strategy. This method builds the suffix array and the BWT block-by-block and in tandem, discarding suffix-array blocks once the corresponding BWT block has been built. By setting a small block size relative to the length of the genome, the technique achieves a modest memory footprint. Also, the algorithm can trade flexibly between speed and peak memory usage by adjusting block size and other parameters. Bowtie's indexer adopts a form of Kärkkäinen's method and can build a full Bowtie index for the human genome in about 24 hours in less than 1.5 gigabytes of RAM. If 16 gigabytes of RAM or more is available, the indexer can exploit the additional RAM to produce the same index in about 4.5 hours.

Physical memory target (gigabytes)	Actual peak memory footprint (gigabytes)	# suffix array blocks	Difference cover period	Bit-packed reference?	Wall clock time
16	14.4	1	256	No	4h:36m
8	5.84	6	1024	No	5h:05m
4	3.39	34	4096	No	7h:40m
2	1.39	34	4096	yes	21h:30m

Table 1: Index-building memory footprint and wall clock time when indexing the whole human genome under various parameters.

Table 1 presents memory footprints and wall clock times for a human-genome run of the indexer under parameters selected to satisfy different physical memory constraints. These runs were performed on a server with a 2.4 GHz AMD Opteron processor and 32 gigabytes of RAM. "Number of blocks" indicates how many blocks the blockwise algorithm used. "Difference cover period" indicates the periodicity of the up-front difference-cover-based pre-sort. This pre-sort, proposed by Burkhardt and Kärkkäinen [59], is a technique whereby a subset of the suffixes are pre-sorted and then used to avoid quadratic runtimes in downstream stages of the indexer. Shorter periods require more up-front work to calculate the pre-sort and more memory to store it, but also yield shorter running times in downstream stages. "2-bit-per-nucleotide references" indicates whether a bit-packed representation of the reference sequence was used. The bit-packed representation reduces memory footprint but increases running time.

Index components

The largest single component of the Bowtie index is the BWT sequence, that is, the sequence of the Burrows-Wheeler-Transformed reference string. Bowtie stores the BWT in a 2-bit-per-nucleotide format. Inline character occurrence counts ("checkpoints") occupy about 14% the space of the packed BWT, and text offsets for marked rows occupy about 50% the space of the packed BWT. With some other small structures, the overall Bowtie index for a given genome is about 65-70% larger than the packed BWT. A Bowtie index for the assembled human genome sequence is about 1.3 gigabytes. As mentioned, a full Bowtie index actually consists of pair of equal-size indexes, the forward and mirror indexes, for any given genome. Bowtie can be run such that only one of the two indexes is ever resident in memory at once (using the -z option), so the memory footprint of Bowtie under those circumstances remains about 1.3 gigabytes. Without the -z option, the human index has a memory footprint of about 2.2 gigabytes, and 2.2 gigabytes is needed to store the index on disk in either case.

Performance results

We evaluated the performance of Bowtie using reads from the 1,000 Genomes project pilot [60]. The reads are stored permanently in the National Center for Biotechnology Information Sequence Read Archive [61] (accession SRR001115). A total of 8.84 million reads, about one "lane" worth of data from an Illumina Genome Analyzer instrument, were trimmed to 35 nt and aligned to the human reference genome (NCBI build 36.3). Unless specified otherwise, read data are not filtered or modified from how they appear in the archive, except in cases where the reads are trimmed prior to the experiment.

All runs were performed on a single processor. Bowtie speedups were calculated as a ratio of wall-clock alignment times. Both wall-clock and CPU times are given to demonstrate that input/output load and competition for the CPU are not significant factors. The time required to build the Bowtie index was not included in the Bowtie running times, since the cost will usually be amortized over many runs, and we anticipate most users will simply download such indices from a public repository. Indices for human, chimp, mouse, dog, rat, and Arabidopsis thaliana genomes (and many others) can be downloaded from the Bowtie website [62].

Results were obtained on two hardware platforms: a desktop workstation with 2.4 GHz Intel Core 2 processor and 2 gigabytes of memory; and a large-memory server with a four-core 2.4 GHz AMD Opteron processor and 32 gigabytes of memory. These are denoted “PC” and “server,” respectively. Both PC and server run Red Hat Enterprise Linux AS release 4.

Comparison to SOAP and MAQ

MAQ [33] is a popular aligner that, at the time of Bowtie’s release, was among the fastest competing open source tools for aligning millions of Illumina reads to the human genome. SOAP [36] is another open source tool that has been reported and used in short-read projects. MAQ and SOAP were the tools used in the very first set of projects that performed sequencing of entire human genomes using sequencing-by-synthesis instruments [63-65].

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (megabytes)	Bowtie speedup	Reads aligned (%)
Bowtie -v 2 (server)	15m:07s	15m:41s	33.8	1,149	-	67.4
SOAP (server)	91h:57m:35s	91h:47m:46s	0.10	13,619	351x	67.3
Bowtie (PC)	16m:41s	17m:57s	29.5	1,353	-	71.9
MAQ (PC)	17h:46m:35s	17h:53m:07s	0.49	804	59.8x	74.7
Bowtie (server)	17m:58s	18m:26s	28.8	1,353	-	71.9
MAQ (server)	32h:56m:53s	32h:58m:39s	0.27	804	107x	74.7

Table 2: Performance results comparing Bowtie to SOAP and MAQ.

Table 2 presents the performance and sensitivity of Bowtie v0.9.6, SOAP v1.10, and MAQ v0.6.6. SOAP could not be run on the PC because SOAP's memory footprint exceeds the PC's physical memory. Bowtie was invoked with the “-v 2” option, which instructs Bowtie to disregard quality values and allow up to 2 mismatches in the alignment, in order to mimic SOAP's default matching policy. Bowtie was invoked with the “--maxns 5” option to mimic SOAP's default policy of filtering out reads with five or more no-confidence positions, where a no-confidence position is usually represented with the letter “N.” For the MAQ comparison Bowtie is run with its default policy, which mimics MAQ's default policy of allowing up to two mismatches in the first 28 nucleotides and enforcing an overall limit of 70 on the sum of the quality values at all mismatched read positions. To make Bowtie's memory footprint more comparable to MAQ's, Bowtie is invoked with the “-z” option in all experiments to ensure that only the forward or mirror index is resident in memory at one time.

SOAP (67.3%) and Bowtie -v 2 (67.4%) are comparable in terms of the fraction of reads aligned. Of the reads aligned by either SOAP or Bowtie, 99.7% were aligned by both, 0.2% were aligned by Bowtie but not SOAP, and 0.1% were aligned by SOAP but not Bowtie. MAQ (74.7%) and Bowtie (71.9%) also align roughly the same fraction of reads, although Bowtie lags by 2.8 percentage points. Of the reads aligned by either MAQ or Bowtie, 96.0% were aligned by both, 0.1% were aligned by Bowtie but not MAQ, and 3.9% were aligned by MAQ but not Bowtie. Of the reads mapped by MAQ but not Bowtie, almost all are due to a flexibility in MAQ's alignment algorithm that allows some alignments to have three mismatches in the seed. The remainder of the reads mapped by MAQ but not Bowtie are due to Bowtie's backtracking ceiling.

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (megabytes)	Bowtie speedup	Reads aligned (%)
Bowtie (PC)	16m:39s	17m:47s	29.8	1,353	-	74.9
MAQ (PC)	11h:15m:58s	11h:22m:02s	0.78	804	38.4x	78.0
Bowtie (server)	18m:20s	18m:46s	28.3	1,352	-	74.9
MAQ (server)	18h:49m:07s	18h:50m:16s	0.47	804	60.2x	78.0

Table 3: Performance results comparing Bowtie to MAQ with filtered reads.

MAQ's documentation mentions that reads containing “poly-A artifacts” can impair MAQ's performance. Table 3 presents performance and sensitivity of Bowtie and MAQ when the read set is filtered using MAQ's “catfilter” command to eliminate poly-A artifacts. The filter eliminates 438,145 out of 8,839,010 reads. Other experimental parameters are identical to those of the experiments in Table 2, and the same observations about the relative sensitivity of Bowtie and MAQ apply here.

Read length and performance

Length	Program	CPU time	Wall clock time	Peak virtual memory footprint (megabytes)	Bowtie speedup	Reads aligned (%)
36 nt	Bowtie	6m:15s	6m:21s	1,305	-	62.2
	MAQ	3h:52m:26s	3h:52m:54s	804	36.7x	65.0
	Bowtie –v 2	4m:55s	5m:00s	1,138	-	55.0
	SOAP	16h:44m:03s	18h:01m:38s	13,619	216x	55.1
50 nt	Bowtie	7m:11s	7m:20s	1,310	-	67.5
	MAQ	2h:39m:56s	2h:40m:09s	804	21.8x	67.9
	Bowtie –v 2	5m:32s	5m:46s	1,138	-	56.2
	SOAP	48h:42m:04s	66h:26m:53s	13,619	691x	56.2
76 nt	Bowtie	18m:58s	19m:06s	1,323	-	44.5
	MAQ 0.7.1	4h:45m:07s	4h:45m:17s	1,155	14.9x	44.9
	Bowtie –v 2	7m:35s	7m:40s	1,138	-	31.7

Table 4: Performance results comparing Bowtie to SOAP and MAQ across read lengths.

Bowtie, MAQ, and SOAP support reads of lengths up to 1,024, 63, and 60 nt, respectively, and MAQ versions 0.7.0 and later support read lengths up to 127 nt. Table 4 shows performance results when the three tools are each used to align three sets of 2 million untrimmed reads, a 36 nt set, a 50 nt set and a 76 nt set, to the human genome on the server platform. Each set of 2 million is randomly sampled from a larger set (from the NCBI Sequence Read Archive: accession SRR003084 for 36 nt, accession SRR003092 for 50 nt, accession SRR003196 for 76 nt). Reads were sampled such that the three sets of 2 million have uniform per-nucleotide error rate, as calculated from per-nucleotide Phred qualities. All reads pass through MAQ's “catfilter.”

Bowtie is run both in its MAQ-like default mode and in its SOAP-like “-v 2” mode. Bowtie is also given the “-z” option to ensure that only the forward or mirror index is resident in memory at one time. MAQ v0.7.1 was used instead of MAQ v0.6.6 for the 76 nt set because v0.6.6 cannot align reads longer than 63 nt. SOAP was not run on the 76 nt set because it does not support reads longer than 60 nt.

The results show that MAQ's algorithm scales better overall to longer read lengths than Bowtie or SOAP. However, Bowtie in SOAP-like “-v 2” mode also scales very well. Bowtie in its default MAQ-like mode scales well from 36 nt to 50 nt reads but is substantially slower for 76 nt reads, although it is still more than an order of magnitude faster than MAQ.

Parallel performance

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (megabytes)	Speedup
Bowtie, 1 thread (server)	18m:19s	18m:46s	28.3	1,353	-
Bowtie, 2 threads (server)	20m:34s	10m:35s	50.1	1,363	1.77x
Bowtie, 4 threads (server)	23m:09s	6m:01s	88.1	1,384	3.12x

Table 5: Performance results when running Bowtie with various numbers of alignment threads.

Alignment can be parallelized by distributing reads across concurrent search threads. Bowtie allows the user to specify a desired number of threads (option -p); Bowtie then launches the specified number of threads using the pthreads library. Bowtie threads synchronize with each other when fetching reads, outputting results, switching between indices, and performing various forms of global bookkeeping. Otherwise, threads are free to operate in parallel, substantially speeding up alignment on computers with multiple processor cores. The memory image of the index is shared by all threads, and so the footprint does not increase substantially when multiple threads are used. Table 5 shows performance results for running Bowtie v0.9.6 on the four-core server with one, two, and four threads.

Bowtie software

Bowtie is an ultrafast, memory-efficient aligner geared toward quickly aligning large sets of short DNA sequences, e.g. 25-75 nucleotides in length, to large genomes such as the human genome. It aligns 35-nucleotide reads to the human genome at a rate of about 25 million reads per hour on a typical workstation. Bowtie indexes the genome with an FM-Index-like structure, which keep sits memory footprint small: an index for the human reference genome fits in less than 3 gigabytes. Multiple processors can be used simultaneously to achieve greater alignment speed. Bowtie can also output alignments in the standard SAM alignment format [66], which allows Bowtie to effectively interoperate with many other tools that support SAM such as the SAMtools SNP caller [66]. Bowtie works best when aligning short reads to large genomes, though it supports arbitrarily small reference sequences (e.g. amplicons) and reads as long as 1024 nucleotides. Bowtie is designed to be extremely fast for sets of short reads where (a) many of the reads have at least one good, valid alignment, (b) many of the reads are relatively high-quality, and (c) the number of alignments reported per read is small (close to 1).

Because of its speed and memory-efficiency, Bowtie has also been used as a subcomponent in many other tools. Tools for RNA sequencing analysis that use Bowtie include TopHat [67], Cufflinks [68], Myrna [69], RNASEQR [70], and GENE-counter [71]. Tools for bisulfite sequencing analysis that use Bowtie include BSmooth [72], Bismark [73], MethylCoder [74] and BS-Seeker [75]. Tools for scaffolding of *de novo* assemblies that use Bowtie include Zorro [76] and SSPACE [77]. General-purpose bioinformatics workflow tools that integrate Bowtie include Galaxy [78] and Chipster [79].

Bowtie is free, open source software available from the Bowtie website at http://bowtie-bio.sf.net.

Chapter 3: Extending Bowtie to finding longer, gapped alignments

This chapter begins by describing two of Bowtie's most significant drawbacks: its poor handling of longer sequencing reads and its inability to find alignments containing gaps. I then describe Bowtie's use of "multiseeding" to support longer reads. I explain why extending Bowtie's search algorithm to hypothesize gaps in addition to mismatches is not an efficient way to support gapped alignment. I go on to describe the whole Bowtie 2 method: a two-stage method that uses the Bowtie index to efficiently find ungapped alignments for seed sequences, then extends seed alignments into full alignments using hardware-accelerated dynamic programming. I then show several performance results.

The design and software implementation of Bowtie 2 are both entirely my work.

Multiseeding: a better seed heuristic for longer reads

While Bowtie (sometimes called "Bowtie 1" here) is an efficient and popular tool, changes in sequencing technology have made its drawbacks apparent. Whereas Bowtie was designed to be efficient and sensitive when aligning short reads (35-50 nt), read lengths produced by second-generation sequencers have increased substantially since Bowtie's release in 2009. Instruments such as Illumina's HiSeq 2000 and GA IIx now routinely produce reads of 100 and 150 nucleotides. Recall the MAQ-like alignment policy discussed in Chapter 2, for example. That policy enforces a stringent ceiling on the amount of dissimilarity permitted in a small "seed" region of the alignment; by default, the seed is the first 28 nucleotides of the read. As reads grow longer, it makes less sense to impose a stringent ceiling on one small portion of the read. Consider a case where the "correct" alignment contains a cluster of edits all falling within the seed region, just barely exceeding the stringent dissimilarity ceiling. Bowtie will not find this alignment even though the overall alignment's percent identity could still be quite high.

Like Bowtie, Bowtie 2 begins the alignment process by imposing a stringent dissimilarity ceiling (e.g. allowing only 0 or 1 mismatches) and aligning a substring of the read (a "seed") subject to that ceiling. Unlike Bowtie, however, Bowtie 2 attempts to align many distinct seeds this way. The seeds are extracted at regular intervals along the read and its reverse complement. Seed strings are contiguous (i.e. they are not spaced seeds) and may or may not overlap each other. For instance, if Bowtie 2 extracts a 20 nt substring every 10 nt along the read, adjacent substrings will overlap by 10 nt. If Bowtie 2 extracts a 18 nt substring every 20 nt, then substrings will not overlap and there will be a gap of 2 nt between adjacent substrings. Figure 14 illustrates how 16 nt seeds are extracted every 10 positions from a read of length 36.

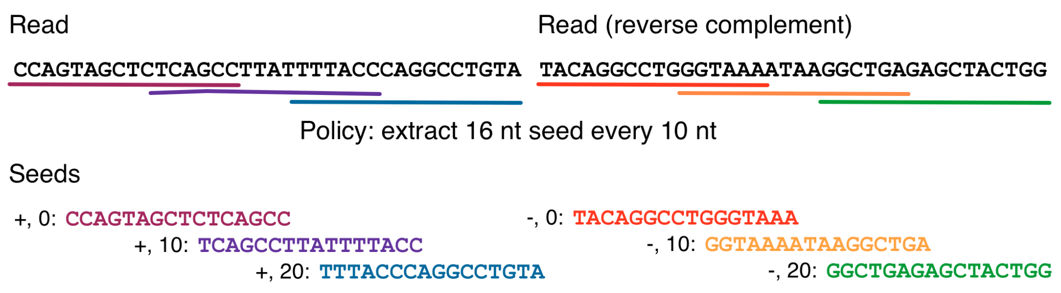

Figure 14: Example of how seeds are extracted in a multiseed scheme.

This is called “multiseeding,” and it is used in Bowtie 2 as a way of “smearing” the stringent dissimilarity ceiling over the entire read, rather than having the ceiling concentrated in one small portion of the read. This makes Bowtie 2 a more appropriate tool for aligning longer, modern sequencing reads.

Support for gapped alignment

Another criticism of Bowtie is that it does not find gapped alignments, i.e., alignments where characters in either the read or reference sequence are skipped over. Gaps in the reference are also called “insertions” and gaps in the read are also called “deletions” (see Figure 15).

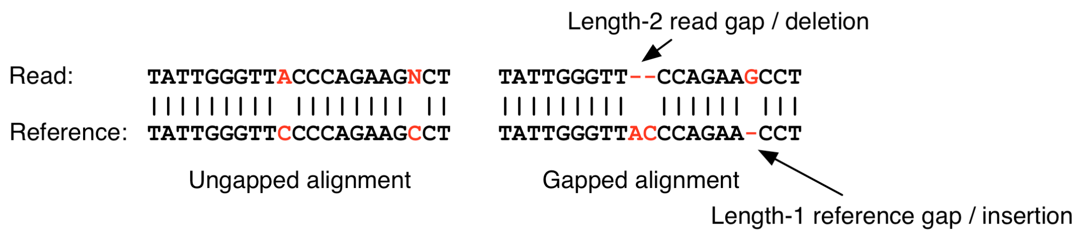

Figure 15: Examples of ungapped and gapped alignments, and gap types.

Insertions and deletions are one common type of mutation observed when comparing two genomes of the same species. For example, a study comparing two finished human reference genomes, each about 3 billion nucleotides long, found on the order of hundreds of thousands of small (1-10 nucleotide) insertions and deletions [11]. As reads grow longer, the probability that a given read overlaps a gap increases, making it important for alignment tools to allow gaps.

Gaps confound pruning

One way to allow gaps is to add the ability to hypothesize gaps directly into Bowtie's usual search procedure. That is, whereas we previously permitted the search to hypothesize either a match or substitution at any given node in the search tree, we might additionally allow the search to hypothesize a gap in the read or a gap in the reference. This is essentially how BWA, a competing alignment tool, handles gaps (see Figure 3 from the BWA study [34]). This increases the number of distinct ways that we might exit a node in the search space from 4 to 9. Of the 9 ways of proceeding, 1 corresponds to a match, 3 correspond to mismatches, 1 corresponds to a reference gap, where a character from the read is placed opposite a gap in the alignment, and 4 correspond to read gaps, where a gap is placed opposite an A, C, G or T in the reference. Another way to state this distinction is that, for ungapped alignment, the maximum outdegree of any search node is 4, whereas in gapped alignment the maximum outdegree is 9. The difference is illustrated in Figure 16.

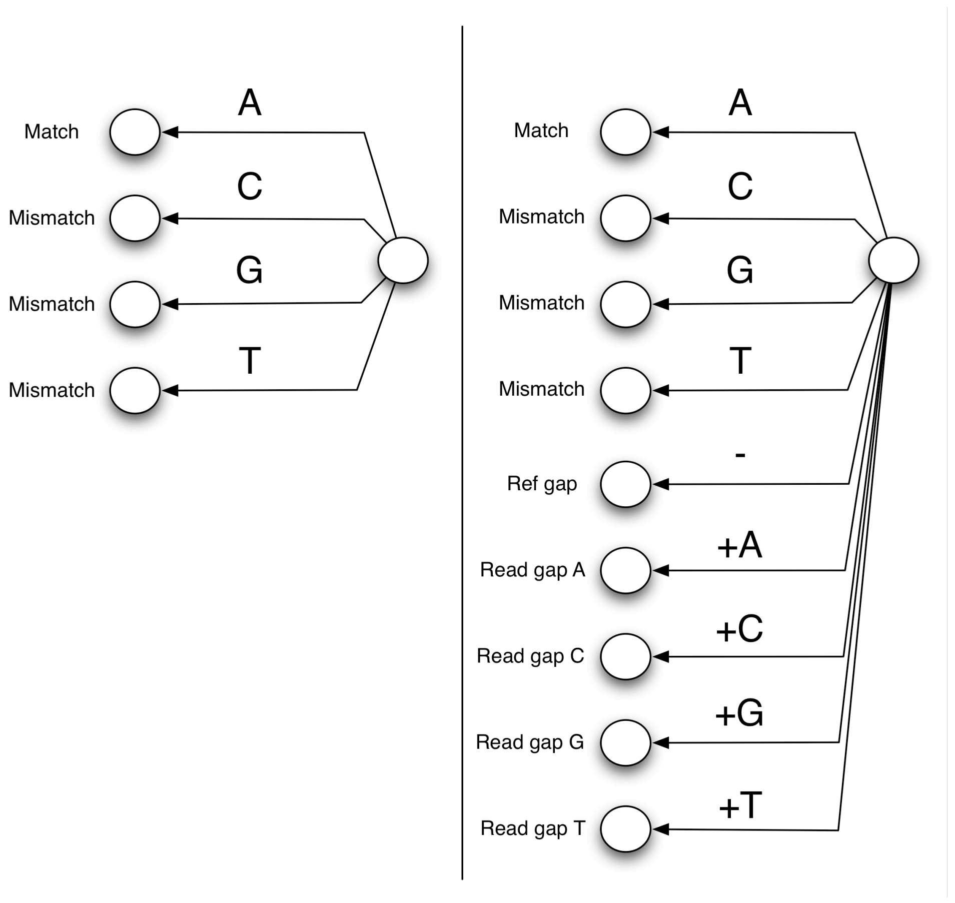

Figure 16: Gaps increase the number of ways we can proceed from a search node.

With the increased outdegree of gapped alignment search nodes, "policy pruning" becomes less effective. That is, there will be less total policy pruning when k edits (gaps or mismatches) are permitted versus when k mismatches are permitted.

A subtler point is that allowing gaps also reduces the effectiveness of reference pruning deep in the search space. Consider an ungapped alignment search where the search has already descended, say, 30 ply down from the root of the search. Assuming the aligned portion of the read is unique, most mismatches we could hypothesize at this depth are pruned by reference pruning. In the case of gapped alignment, however, many potential gaps are not eliminated by reference pruning. A reference gap (insertion), for instance, does not introduce any additional characters to the reference side of the alignment, and so reference pruning does not apply. A read gap (deletion) does introduce characters on the reference side of the alignment and therefore can be pruned by reference pruning, but it is always possible to construct a read gap that passes reference pruning by building the gap from characters occurring the left of the already-matched reference characters.

On the whole, allowing gaps greatly reduces the effectiveness of Bowtie's pruning strategies, both near and far from the root of the search space. The situation is worse when more and longer gaps are permitted, and therefore will also generally be worse when aligning longer reads. For these reason, Bowtie 2 chooses a different strategy for finding gapped alignments.

A new division of labor

Bowtie 1 uses an FM-Index-assisted search strategy to find ungapped read alignments. The strategy is efficient largely due to pruning, which eliminates a large fraction of the space. However, pruning becomes far less effective when gaps are permitted. Another gapped alignment strategy is to use dynamic programming, e.g. Needleman-Wunsch [80], Smith-Waterman [31], or extensions thereof [32]. Dynamic programming scales well with the number and length of gaps permitted. In fact, for a fixed-size dynamic programming matrix, allowing more gaps and longer gaps has no effect on the worst-case performance and little to no effect on average-case performance. However, as we have also seen, dynamic programming is extremely slow when used to align a read against an entire, long reference sequence. In summary: while FM-Index-assisted search is quite efficient for finding short, ungapped alignments, it slows substantially when more and longer gaps are permitted; dynamic programming, on the other hand, is very slow when used to align to the whole genome, but very robust to increases in the number and length of gaps permitted.

Bowtie 2 attempts to combine the strengths of these approaches while avoiding most of their weaknesses. In Bowtie 2, FM-Index-assisted search is used to find short, ungapped "seed alignments," i.e. alignments for short substrings extracted from the read, rather than full alignments. Dynamic programming is then used to extend seed alignments into full, gapped alignments; it is not used to align to the entire reference genome. Bowtie 2 accelerates dynamic programming using Single Instruction Multiple Data ("SIMD") instructions (also called "vector" or "streaming" instructions) available on all modern processors. This division of labor between FM-Index-assisted ungapped alignment and SIMD-accelerated dynamic programming plays to strengths of both approaches and sidesteps their biggest weaknesses.

Bowtie 2 workflow

When aligning an unpaired read, Bowtie 2 proceeds in four steps. In step 1, Bowtie 2 extracts substrings ("seed" strings) from the read and its reverse complement. In step 2, the seed strings are aligned to the genome in an ungapped fashion with the aid of the FM Index. In step 3, seed alignments are prioritized and their offsets with respect to the reference genome are determined. Step 4 takes prioritized, resolved alignments from step 3 and performs SIMD-accelerated dynamic programming alignment in the vicinity of each until all are examined, until a sufficient number of alignments were examined, or until another limit is reached. These steps are illustrated in Figure 17 and described in greater detail in the following subsections.

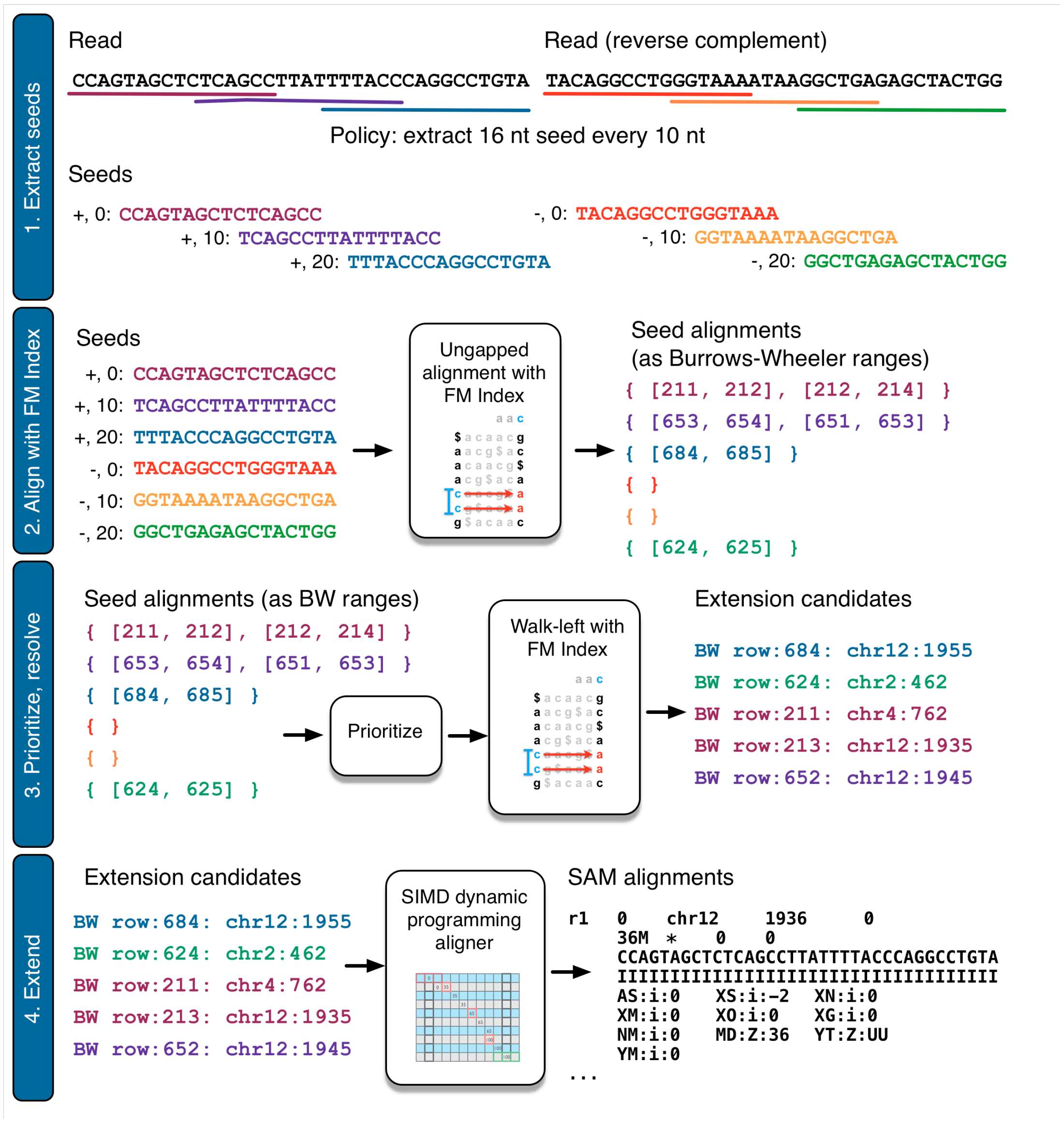

Figure 17: Steps of the Bowtie 2 workflow

Seed extraction

Substrings of the read (“seed strings”) are extracted at regular intervals along the read and its reverse complement, per the multiseeding policy discussed previously. Seed length is configured using Bowtie 2’s -L option. The -L option can take any value from 4 through 32. Values for this option that performed well in our experiments ranged from 20 to 25. When the input comprises reads of various lengths (e.g. for 454 or Ion Torrent data), it is advantageous to set vary interval length from read to length using a sublinear function of read length. For instance, the default function used in Bowtie 2 end-to-end mode is:

$$I(x) = \lfloor 1 + 1.15\sqrt{x} \rfloor \tag{3.1}$$

I is interval length as a function of the length of the read, x. For a 100 nt read, this causes seeds to start 12 nt apart, with the first seed starting at offset 0 from the left (5’) end, the second seed starting at offset 12, etc. The constant term (1 in this case), coefficient (1.15 in this case) and function used (square root in this case) are configurable via Bowtie 2’s -i option. Other functions that can be used include: a constant function (equivalent to specifying a coefficient of 0), a linear function of x, or log(x). In my experiments using data from a 454 sequencer, square root seemed to provide the most advantageous combination of speed and fraction of reads aligned.

FM Index-assisted seed alignment

Given seed strings, Bowtie 2 then uses FM Index-assisted alignment to find ungapped alignments for each seed. The alignment process makes use of the same reference pruning, policy pruning and double indexing approaches used in Bowtie

[18]. In addition, Bowtie 2 uses the bi-directional BWT [55] approach, which allows the aligner to efficiently switch between alignment in a right-to-left direction and alignment in a left-to-right direction.

Seed strings can be aligned with up to 1 mismatch. The number of mismatches to permit is configurable. Option -N 1 allows seed alignments to have up to 1 mismatch, whereas option -N 0 requires that seeds match exactly.

Seed alignment prioritization

The output from the seed alignment step is a set of zero or more Burrows-Wheeler ranges per seed string. Such a range is called a "seed-hit range." A seed-hit range describes a range of rows in the Burrows-Wheeler matrix that begin with a reference substring that is within 0 or 1 mismatches of the seed substring. A single seed string may be associated with multiple Burrows-Wheeler ranges, since a seed string may be within 1 mismatch of many distinct reference substrings. Each row of each seed-hit range corresponds to a location in the reference genome where we might search for a full alignment. Bowtie 2 assigns a weight to each Burrows-Wheeler row equal to $1 / r^2$ where r is the total number of rows in the range. E.g. a row from a seed-hit range with 3 elements gets 1 / 9th the weight of a row from a seed-hit range with 1 element.

In this step, Bowtie 2 proceeds by repeatedly selecting a row in a random weighted fashion using these weights. When a row is selected, its offset into the reference genome is resolved using the same hybrid approach as Bowtie. Each resolved offset is passed to the SIMD-accelerated dynamic programming algorithm along with information about which seed string gave rise to the hit.

SIMD-accelerated dynamic programming

For each resolved seed hit, Bowtie 2 extracts flanking characters from the reference and solves a rectangular dynamic programming problem to find high-scoring full alignments in the vicinity of the seed hit. Dynamic programming alignment algorithms such as Needleman-Wunsch [81], Smith-Waterman [31], and extensions thereof [32], enable efficient computation of the optimal alignment between two sequences, even in the presence of many gaps and mismatches.

Dynamic programming algorithms can be visualized as acting on a matrix with rows corresponding to characters in the read and columns corresponding to characters in the reference. The algorithm calculates ("fills") all elements in the matrix moving from the upper left corner to the lower right corner, with each element (i, j) set to the alignment score that results from aligning the length-i prefix of the read to the length-j prefix of the reference. Because a given cell (i, j) can be calculated by considering only values in the cells above (i-1, j), to the left (i, j-1) and to the upper-left (i-1, j-1), dynamic programming can be parallelized. For instance, consider a matrix for which all the elements in the first N anti-diagonals have already been calculated. All of the elements in the N+1th anti-diagonal can be calculated simultaneously in parallel. That is, the inputs for anti-diagonal N are available in previous anti-diagonals and none of the calculations for anti-diagonal N depend on each other. **Figure 18** shows an example where the first 5 anti-diagonals have already been calculated, and all the elements in the 6th anti-diagonal, highlighted in read, can be calculated in parallel.

	C	G	T	T	T	A	C
C	1	0	0	0	0	🟥	
G	0	2	0	0	🟥		
T	0	1	3	🟥			
T	0	0	🟥				
A	0	🟥					
C	🟥						

Figure 18: Highlighted cells can be filled in parallel.

Many approaches for accelerating dynamic programming have been proposed, including implementations that use single-instruction multiple-data (SIMD) instructions (also called "vector" or "streaming" instructions) available on general purpose CPUs. SIMD instructions are similar to normal processor instructions, in that they perform simple arithmetic and logical functions such as addition, multiplication, maximum, logical AND, etc. However, instead of operating on two operands (A+B), SIMD instructions operate on many operands in parallel (A+B and C+D and E+F, etc.). The operands are packed into long SIMD machine words and the operations are carried out using a special SIMD arithmetic unit. On modern processors, these are called SSE ("Streaming SIMD Extensions") instructions. Certain standard sets of SSE instructions have been supported on both Intel and AMD processors for many years now.

Bowtie 2 chiefly makes use of SSE2 instructions, which include a standard set of arithmetic and logic functions operating over 128-bit words. In Bowtie 2, two different ways of "packing" values into 128-bit words are used: either 16 unsigned 8-bit values are packed into a single word, or 8 16-bit signed values are packed into a single word. Figure 19 Illustrates an example where two 128-words with 8 16-bit values packed into each are added together element-wise.

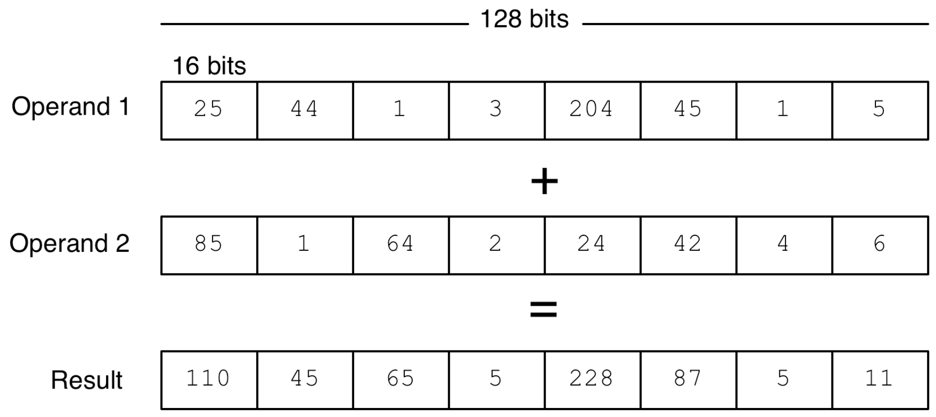

Figure 19: SIMD addition of two vectors of 8 16-bit values each

One of the first approaches for accelerating dynamic programming using SIMD instructions was proposed by Wozniak [82] in 1996. The approach builds on the insight illustrated in **Figure 18**, namely that cells on an anti-diagonal are independent and can be computed in parallel provided that previous anti-diagonals have already been calculated. Since then, a series of publications, including by Rognes and Seeberg [83], Farrar [84], and Rognes [85], showed other ways of parallelizing the problem, including approaches that fill the matrix using horizontal or vertical blocks.

Bowtie 2 builds on the approach used by the swsse2 tool, published by Farrar [84], in which the matrix is filled in one striped, vertical chunk at a time. Because the chunks are oriented vertically, the read can be preprocessed into a "query profile," and diagonal score contributions can be calculated using a lookup table. Because the lookup table depends only on the read, it can be reused across dynamic programming problems involving the read. Because the chunks are "striped" along the read, the work required to propagate vertical contributions is reduced compared to non-striped approaches.

While the swsse2 tool is geared toward scoring protein alignments, Bowtie 2 adapts and extends the approach for read alignment. Specifically, Bowtie 2's approach (a) works for end-to-end alignment in addition to local alignment, (b) implements a restriction on which positions may contain gaps, (c) implements separately configurable read and reference gap penalties, (d) permits scoring functions that account for quality values, and (e) implements a backtrack procedure so that alignments can be derived directly from the algorithm's output.

Performance comparison on real data

To assess how Bowtie 2 performs on real-world data, Bowtie 2 v2.0.0-beta4 was compared to three other FM-Index-based read aligners: BWA 0.5.9-r16 [34], BWA-SW 0.5.9-r16 [35], and SOAP2 2.21 [37]. In all experiments, the reference used was the GRCh37 major build of the human genome [86], including sex chromosomes, mitochondrial genome and "non-chromosomal" sequences. An illustrative plot of all the results in this subsection is shown in Figure 20 and full results are found in Table 6.

A random subset of 2 million pairs was extracted from a collection of 100-by-100 nt paired-end HiSeq 2000 reads from a human DNA sequencing study [87] (accession ERR037900). BWA, SOAP2 and Bowtie 2 were used to align one end (labeled "1") from the subset in an unpaired fashion. To illustrate parameter tradeoffs, each tool was tool with a variety of parameter settings (Figure 20a, Table 6a). Note that SOAP2 does not permit gapped alignment of unpaired reads. Bowtie 2's default mode (labeled 3 in Figure 20a) is faster than all BWA modes tried and more than 2.5 times faster than BWA's default mode (labeled 7 in Figure 20a). All Bowtie 2 modes yielded a greater number of reads aligned than did either of the other tools. Bowtie 2's peak virtual memory footprint (3.24 gigabytes) was between BWA's (2.39 gigabytes) and SOAP2's (5.34 gigabytes).

BWA, SOAP2 and Bowtie 2 were also used to align reads from this dataset in a paired-end fashion, using various alignment parameters (Figure 20b, Table 6b). Bowtie 2's default mode (labeled 3 in Figure 20b) is faster than all BWA modes tried and more than 3 times faster than BWA's default mode (labeled 7 in Figure 20b). All Bowtie 2 modes yielded a greater number of reads aligned than did either of the other tools. Bowtie 2's peak virtual memory footprint (3.26 gigabytes) was similar to BWA's (3.20 gigabytes) and less than SOAP2's (5.34 gigabytes).

To assess Bowtie 2's performance on longer reads, a random subset of 1 million reads were extracted from both (a) a collection of 454 reads from the 1000 Genomes Project Pilot [60] (accession SRR003161), and (b) a collection of Ion Torrent reads from the G. Moore DNA sequencing project [88] (accession ERR039480). The subset was aligned with BWA-SW and Bowtie 2. Bowtie 2 was configured to perform local alignment, which is also BWA-SW's behavior. Both tools were run with various parameter settings. For the 454 data (Figure 20c, Table 6c), Bowtie 2's default local-alignment mode (labeled 3 in Figure 20c) was faster and aligned more reads than any of the BWA-SW modes tried. In the case of the Ion Torrent data (Figure 20d, Table 6d), Bowtie 2's default local-alignment mode (labeled 3 in Figure 20d) was more than twice as fast as BWA-SW's default mode (labeled 7 in Figure 20d), and aligns more reads. For both the 454 and Ion Torrent datasets, Bowtie 2's peak virtual memory footprint (3.39 gigabytes) was smaller than BWA-SW's (3.66 gigabytes).

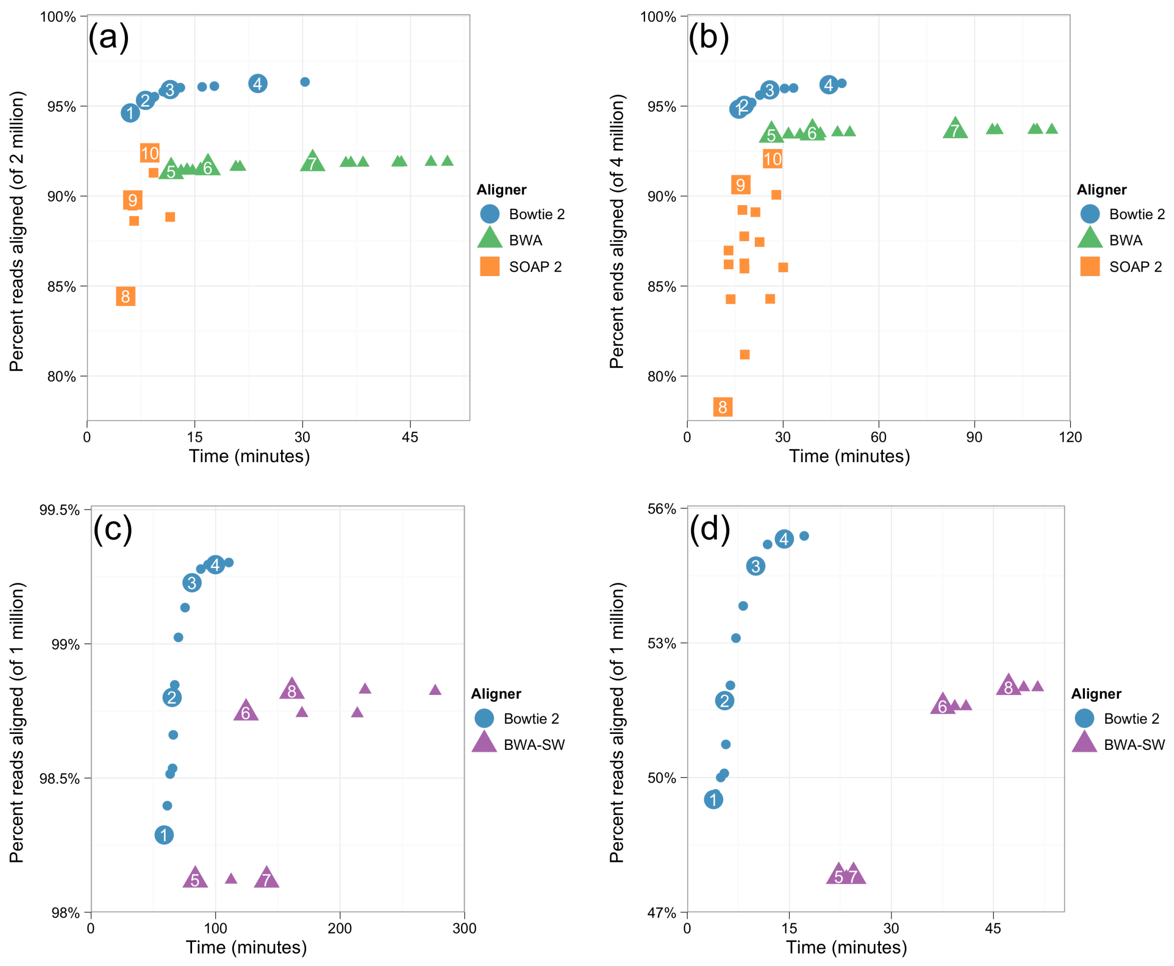

Figure 20: Speed and percent reads aligned for Bowtie 2 versus others.

Aligner	Options	Label in Figure 20	Running time	% reads aligned (of 2 million)	Peak virtual memory footprint (gigabytes)
(a) Unpaired 100 nt HiSeq 2000 data					
Bowtie 2	-D 5 -R 1 -N 0 -L 22 -i S,0,2.50 (--very-fast)	1	6m:02s	94.62%	3.24
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,0,2.50 (--fast)	2	8m:08s	95.32%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.50		9m:15s	95.49%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.20		9m:23s	95.52%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.65		10m:36s	95.80%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.15 (--sensitive)	3	11m:32s	95.92%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 21 -i S,1,1.00		12m:59s	96.03%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.75		16m:01s	96.07%	3.24
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.50		17m:43s	96.11%	3.24
Bowtie 2	-D 20 -R 3 -N 0 -L 20 -i S,1,0.50 (--very-sensitive)	4	23m:44s	96.26%	3.24
Bowtie 2	-D 25 -R 4 -N 0 -L 20 -i S,1,0.50		30m:20s	96.34%	3.24

BWA	-k 1 -l 32 -o 1	5	11m:42s	91.36%	2.39
BWA	-k 1 -l 32 -o 2		13m:05s	91.40%	2.44
BWA	-k 1 -l 28 -o 1		13m:55s	91.47%	2.40
BWA	-k 1 -l 32 -o 3		14m:41s	91.40%	2.52
BWA	-k 1 -l 28 -o 2		15m:48s	91.51%	2.48
BWA	-k 1 -l 24 -o 1	6	16m:50s	91.57%	2.41
BWA	-k 1 -l 28 -o 3		17m:25s	91.51%	2.56
BWA	-k 1 -l 24 -o 2		20m:42s	91.61%	2.51
BWA	-k 1 -l 24 -o 3		21m:17s	91.61%	2.59
BWA	-k 2 -l 32 -o 1	7	31m:24s	91.80%	2.41
BWA	-k 2 -l 28 -o 1		36m:01s	91.83%	2.41
BWA	-k 2 -l 32 -o 2		36m:43s	91.84%	2.51
BWA	-k 2 -l 32 -o 3		38m:25s	91.84%	2.59
BWA	-k 2 -l 28 -o 2		43m:13s	91.87%	2.52
BWA	-k 2 -l 24 -o 1		43m:17s	91.85%	2.42
BWA	-k 2 -l 28 -o 3		43m:44s	91.87%	2.59
BWA	-k 2 -l 24 -o 2		47m:52s	91.89%	2.53
BWA	-k 2 -l 24 -o 3		50m:09s	91.89%	2.63
SOAP2	-l 256 -v 3 -g 0		5m:20s	84.43%	5.34
SOAP2	-l 256 -v 5 -g 0	8	5m:23s	84.43%	5.34
SOAP2	-l 256 -v 7 -g 0		5m:30s	84.43%	5.34
SOAP2	-l 75 -v 5 -g 0		6m:20s	89.47%	5.34
SOAP2	-l 75 -v 7 -g 0	9	6m:22s	89.78%	5.34
SOAP2	-l 75 -v 3 -g 0		6m:33s	88.62%	5.34
SOAP2	-l 40 -v 7 -g 0	10	8m:44s	92.40%	5.34
SOAP2	-l 40 -v 5 -g 0		9m:15s	91.29%	5.34
SOAP2	-l 40 -v 3 -g 0		11m:34s	88.84%	5.34

(b) Paired-end 100 x 100 nt HiSeq 2000 data

Bowtie 2	-D 5 -R 1 -N 0 -L 22 -i S,0,2.50 (--very-fast)	1	16m:09s	94.80%	3.25
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,0,2.50 (--fast)	2	17m:51s	95.04%	3.25
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.50		20m:08s	95.16%	3.25
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.20		20m:09s	95.20%	3.25
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.65		22m:43s	95.61%	3.26
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.15 (--sensitive)	3	25m:52s	95.90%	3.26
Bowtie 2	-D 15 -R 2 -N 0 -L 21 -i S,1,1.00		27m:16s	95.92%	3.26
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.75		30m:28s	95.98%	3.26
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.50		33m:16s	96.00%	3.26
Bowtie 2	-D 20 -R 3 -N 0 -L 20 -i S,1,0.50 (--very-sensitive)	4	44m:09s	96.19%	3.26
Bowtie 2	-D 25 -R 4 -N 0 -L 20 -i S,1,0.50		48m:24s	96.27%	3.26
BWA	-k 1 -l 32 -o 1	5	26m:23s	93.38%	3.20
BWA	-k 1 -l 32 -o 2		31m:39s	93.41%	3.20
BWA	-k 1 -l 28 -o 1		31m:40s	93.45%	3.20
BWA	-k 1 -l 32 -o 3		35m:17s	93.41%	3.20
BWA	-k 1 -l 28 -o 2		38m:03s	93.47%	3.20
BWA	-k 1 -l 24 -o 1	6	39m:11s	93.51%	3.20

BWA	-k 1 -l 28 -o 3		41m:40s	93.47%	3.20
BWA	-k 1 -l 24 -o 2		47m:02s	93.53%	3.20
BWA	-k 1 -l 24 -o 3		50m:53s	93.53%	3.20
BWA	-k 2 -l 32 -o 1	7	83m:58s	93.63%	3.20
BWA	-k 2 -l 32 -o 2		95m:28s	93.65%	3.20
BWA	-k 2 -l 28 -o 1		95m:35s	93.65%	3.20
BWA	-k 2 -l 32 -o 3		97m:04s	93.66%	3.20
BWA	-k 2 -l 24 -o 1		108m:30s	93.66%	3.20
BWA	-k 2 -l 28 -o 2		109m:31s	93.67%	3.20
BWA	-k 2 -l 28 -o 3		114m:09s	93.67%	3.20
BWA	-k 2 -l 24 -o 2		127m:20s	93.68%	3.20
BWA	-k 2 -l 24 -o 3		131m:48s	93.68%	3.20
SOAP2	-l 256 -v 3 -g 0 -m 250 -x 500		11m:10s	78.28%	5.34
SOAP2	-l 256 -v 5 -g 0 -m 250 -x 500	8	11m:10s	78.28%	5.34
SOAP2	-l 256 -v 7 -g 0 -m 250 -x 500		11m:14s	78.28%	5.34
SOAP2	-l 75 -v 7 -g 0 -m 250 -x 500		12m:55s	86.97%	5.35
SOAP2	-l 75 -v 5 -g 0 -m 250 -x 500		12m:56s	86.20%	5.35
SOAP2	-l 75 -v 3 -g 0 -m 250 -x 500		13m:34s	84.27%	5.35
SOAP2	-l 75 -v 7 -g 3 -m 250 -x 500	9	16m:48s	90.63%	5.35
SOAP2	-l 75 -v 5 -g 3 -m 250 -x 500		17m:15s	89.23%	5.35
SOAP2	-l 256 -v 7 -g 3 -m 250 -x 500		17m:50s	87.76%	5.35
SOAP2	-l 256 -v 5 -g 3 -m 250 -x 500		17m:51s	86.26%	5.35
SOAP2	-l 75 -v 3 -g 3 -m 250 -x 500		17m:54s	85.97%	5.35
SOAP2	-l 256 -v 3 -g 3 -m 250 -x 500		18m:01s	81.19%	5.36
SOAP2	-l 40 -v 7 -g 0 -m 250 -x 500		21m:20s	89.11%	5.35
SOAP2	-l 40 -v 5 -g 0 -m 250 -x 500		22m:38s	87.45%	5.35
SOAP2	-l 40 -v 3 -g 0 -m 250 -x 500		25m:57s	84.29%	5.35
SOAP2	-l 40 -v 7 -g 3 -m 250 -x 500	10	26m:43s	92.08%	5.35
SOAP2	-l 40 -v 5 -g 3 -m 250 -x 500		27m:53s	90.07%	5.35
SOAP2	-l 40 -v 3 -g 3 -m 250 -x 500		30m:02s	86.04%	5.35
(c) 454 data					
Bowtie 2	-D 5 -R 1 -N 0 -L 25 -i S,1,2.0 --bwa-sw-like	1	58m:41s	98.29%	3.27
Bowtie 2	-D 5 -R 1 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		61m:12s	98.40%	3.27
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		63m:28s	98.51%	3.27
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,1,1.75 --bwa-sw-like	2	65m:05s	98.80%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		65m:25s	98.54%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.20 --bwa-sw-like		65m:59s	98.66%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.65 --bwa-sw-like		67m:09s	98.85%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.15 --bwa-sw-like		70m:08s	99.02%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 21 -i S,1,1.00 --bwa-sw-like		75m:28s	99.13%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.75 --bwa-sw-like	3	82m:14s	99.23%	3.27
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like		88m:06s	99.28%	3.28
Bowtie 2	-D 20 -R 3 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like	4	93m:55s	99.29%	3.28
Bowtie 2	-D 25 -R 4 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like		110m:39s	99.30%	3.28
BWA-SW	-c 5.5 -z 1 -s 1	5	83m:41s	98.12%	3.66

BWA-SW	-c 5.5 -z 1 -s 2		112m:26s	98.12%	3.66
BWA-SW	-c 5.5 -z 2 -s 1	6	124m:28s	98.74%	3.68
BWA-SW	-c 5.5 -z 1 -s 3	7	141m:00s	98.12%	3.66
BWA-SW	-c 5.5 -z 3 -s 1	8	161m:21s	98.82%	3.69
BWA-SW	-c 5.5 -z 2 -s 2		169m:20s	98.74%	3.68
BWA-SW	-c 5.5 -z 2 -s 3		213m:47s	98.74%	3.68
BWA-SW	-c 5.5 -z 3 -s 2		220m:01s	98.83%	3.69
BWA-SW	-c 5.5 -z 3 -s 3		276m:22s	98.82%	3.69
(d) Ion Torrent data					
Bowtie 2	-D 5 -R 1 -N 0 -L 25 -i S,1,2.0 --bwa-sw-like	1	3m:52s	49.51%	3.37
Bowtie 2	-D 5 -R 1 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		4m:10s	49.64%	3.37
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		4m:55s	50.00%	3.37
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.50 --bwa-sw-like		5m:26s	50.09%	3.37
Bowtie 2	-D 10 -R 2 -N 0 -L 22 -i S,1,1.75 --bwa-sw-like	2	5m:30s	51.72%	3.37
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,2.20 --bwa-sw-like		5m:40s	50.74%	3.37
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.65 --bwa-sw-like		6m:20s	52.05%	3.37
Bowtie 2	-D 15 -R 2 -N 0 -L 22 -i S,1,1.15 --bwa-sw-like		7m:09s	53.11%	3.37
Bowtie 2	-D 15 -R 2 -N 0 -L 21 -i S,1,1.00 --bwa-sw-like		8m:13s	53.82%	3.38
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.75 --bwa-sw-like	3	10m:00s	54.71%	3.38
Bowtie 2	-D 15 -R 2 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like		11m:48s	55.19%	3.39
Bowtie 2	-D 20 -R 3 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like	4	14m:20s	55.32%	3.39
Bowtie 2	-D 25 -R 4 -N 0 -L 20 -i S,1,0.50 --bwa-sw-like		17m:11s	55.38%	3.39
BWA-SW	-c 5.5 -z 1 -s 1	5	22m:16s	47.80%	3.66
BWA-SW	-c 5.5 -z 1 -s 2		23m:23s	47.80%	3.66
BWA-SW	-c 5.5 -z 1 -s 3	7	24m:26s	47.80%	3.66
BWA-SW	-c 5.5 -z 2 -s 1	6	37m:34s	51.58%	3.67
BWA-SW	-c 5.5 -z 2 -s 2		39m:19s	51.58%	3.67
BWA-SW	-c 5.5 -z 2 -s 3		40m:58s	51.58%	3.67
BWA-SW	-c 5.5 -z 3 -s 1	8	47m:14s	52.01%	3.67
BWA-SW	-c 5.5 -z 3 -s 2		49m:27s	52.01%	3.67
BWA-SW	-c 5.5 -z 3 -s 3		51m:30s	52.01%	3.67

Table 6: Speed and percent reads aligned for Bowtie 2 versus others: full results.

Accuracy and sensitivity comparison on simulated data

To assess accuracy and sensitivity of Bowtie 2, a series of studies were conducted using simulated sequencing reads. Mason [89] was used to simulate sets of 100,000 Illumina-like reads 100 nt long and 150 nt long from the genome. Similarly, Mason was used to simulated datasets of 100,000 paired-end reads of lengths 100 x

100 nt and 150 x 150 nt. Bowtie 2, BWA, and SOAP2 were run with their default arguments on each dataset. For each run, the number of reads aligned correctly and incorrectly were counted. An alignment was considered correct if the strand was correct, and if the leftmost position covered by the alignment was within 50 nt of the leftmost position chosen by the simulator. Otherwise the alignment was considered incorrect. For each aligner and each dataset, correct and incorrect alignments were tallied, stratified by mapping quality. Mapping quality is defined as -10 log10(p) rounded to the nearest integer where p is the aligner's estimate of the probability that the read was aligned incorrectly. The cumulative number of correct and incorrect alignments were counted and accumulated from high to low mapping quality. Figure 21a, Figure 21b show a plot of cumulative correct alignments on the vertical axis and cumulative incorrect alignments on the horizontal axis for each dataset and aligner. In all cases, Bowtie 2 and BWA report more correct alignments than SOAP2. For the unpaired Illumina-like reads, Bowtie 2's plotted curve is above BWA's, indicating Bowtie 2 reports more correct alignments and fewer incorrect alignments over a range of mapping quality cutoffs. For paired-end reads, the difference is smaller but Bowtie 2's curve lies mostly above BWA's. Note that, in its paired-end mode, BWA performs local alignment to recover one of the two ends of the paired-end read in some situations, which Bowtie 2 does not. In the 150 nt comparison, for instance, BWA trims 2,991 reads in this way.

The Mason simulator was also used to generate two collections of 100,000 454-like reads with average lengths 250 and 400 respectively. Bowtie 2 and BWA-SW were run on these datasets (Figure 21c). Bowtie 2's curve is generally above

BWA-SWs, especially for the dataset with average length 250. BWA-SW also trims in more cases. For the dataset with average length 250, for example, BWA-SW trims 53,486 reads while Bowtie 2 trims 51,051.

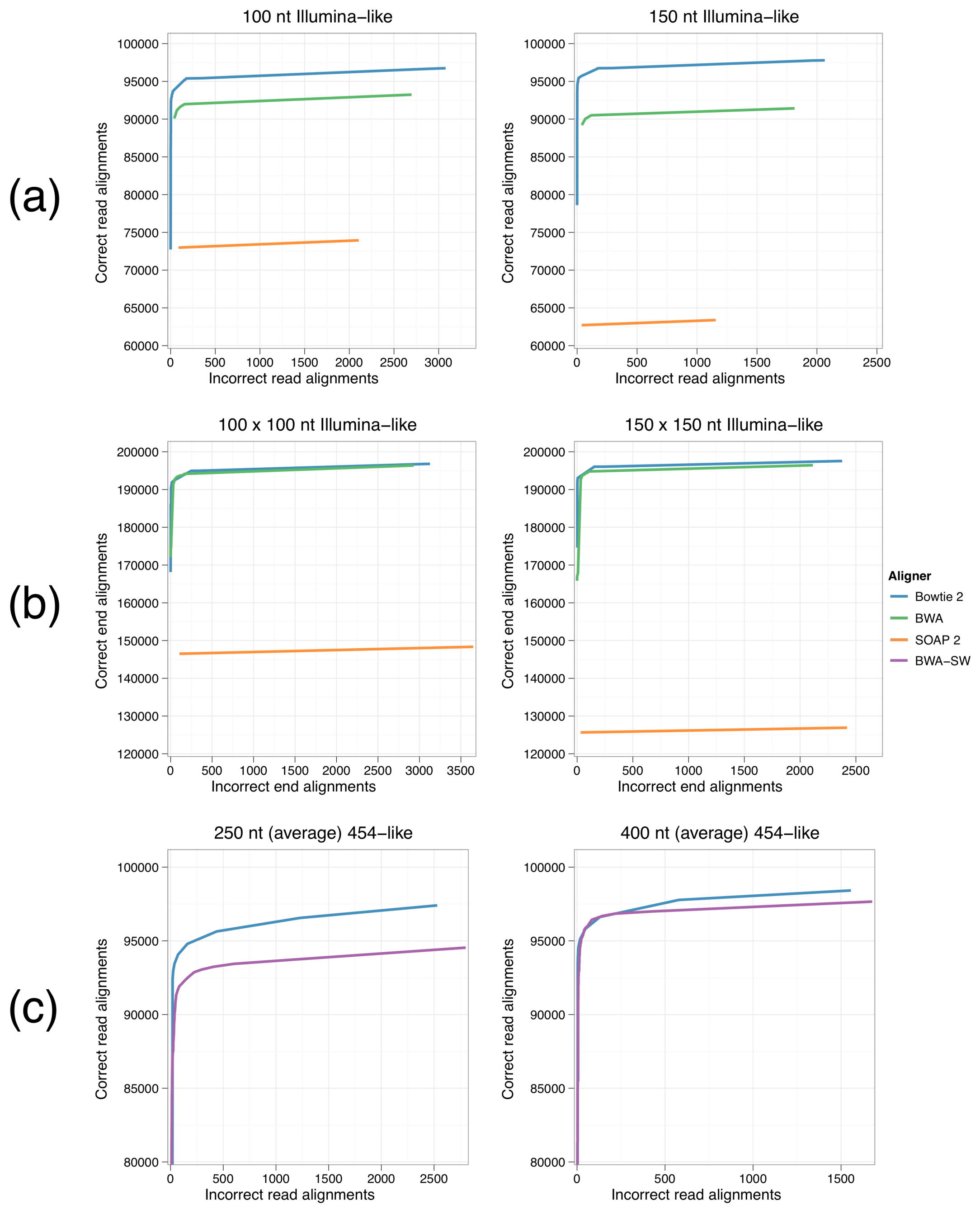

Figure 21: Alignment ROC curves for Bowtie 2 and other tools

Comparison of Bowtie 1 and Bowtie 2

To see how Bowtie 1 and Bowtie 2 compare in terms of speed and fraction of reads aligned, Bowtie 2 and Bowtie 1 were run on the same set of 100 x 100 nt reads used in the comparison illustrated in Figure 20. Bowtie 1 was designed to align relatively short reads (i.e. shorter than 100 nt), so here Bowtie 1 and 2 are compared for both unpaired (Figure 22, Table 7) and paired-end reads (Figure 23, Table 8) of length 40, 60, 80 and 100 nt. The 40, 60, and 80 nt datasets were constructed by trimming bases from the right (3') end of the 100 nt dataset.

The minimum and maximum insert lengths were set to 0 and 500 for both tools. Bowtie 2 was run in its default mode. Bowtie 1's reporting options were set to be as comparable as possible to Bowtie 2's defaults ("-M 1 --best"). Bowtie 1 was run in "-v 2" mode, which allows up to 2 mismatches in the entire alignment. Bowtie 1 was also run in "-l 28 -n 2" mode, which uses the first 28 nt of the read as a "seed" and allows at most 2 mismatches in that portion. The "-e" option sets a ceiling on the sum of the quality scores at mismatched positions, where quality scores are rounded to the nearest 10 and scores greater than 30 are rounded to 30. Two settings for "-e" were used: 100, and 250.

Note that Bowtie 2 will attempt to find and report alignments for each end separately if the ends cannot be aligned concordantly as a pair. Bowtie 1, on the other hand, reports no alignment for either end in this case. Thus, the paired-end comparison (Figure 23, Table 8) lends a small speed advantage to Bowtie 1 and a sensitivity advantage to Bowtie 2.

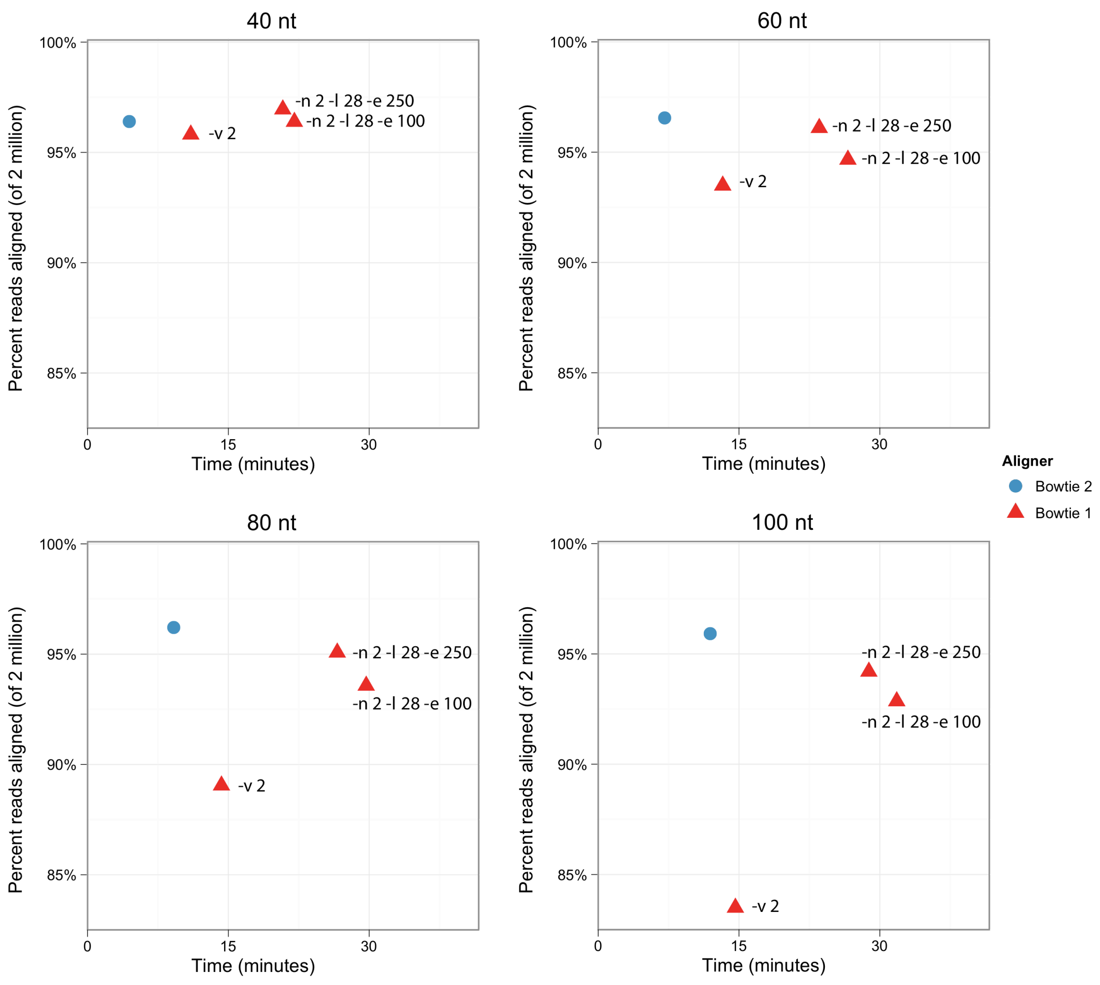

Figure 22: Comparison of Bowtie 1 and Bowtie 2 on unpaired reads.

Aligner	Options	Running time	% reads aligned (out of 2 million)	Peak virtual memory footprint (gigabytes)
Length: 40 nt				
Bowtie 2	(defaults)	4m:27s	96.40%	3.35
Bowtie 1	-v 2 -M 1 --best	11m:00s	95.81%	2.34
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best	22m:02s	96.39%	2.34
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best	20m:48s	96.95%	2.34
Length: 60 nt				
Bowtie 2	(defaults)	6m:09s	96.55%	3.24
Bowtie 1	-v 2 -M 1 --best	13m:16s	93.49%	2.34
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best	26m:36s	94.66%	2.34
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best	23m:33s	96.10%	2.34
Length: 80 nt				
Bowtie 2	(defaults)	9m:11s	96.21%	3.24
Bowtie 1	-v 2 -M 1 --best	14m:16s	89.05%	2.34
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best	29m:41s	93.57%	2.34
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best	26m:36s	95.07%	2.34
Length: 100 nt				
Bowtie 2	(defaults)	11m:56s	95.92%	3.24
Bowtie 1	-v 2 -M 1 --best	14m:37s	83.50%	2.34
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best	31m:48s	92.86%	2.34
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best	28m:50s	94.20%	2.34

Table 7: Comparison of Bowtie 1 and Bowtie 2 on unpaired reads: full results.

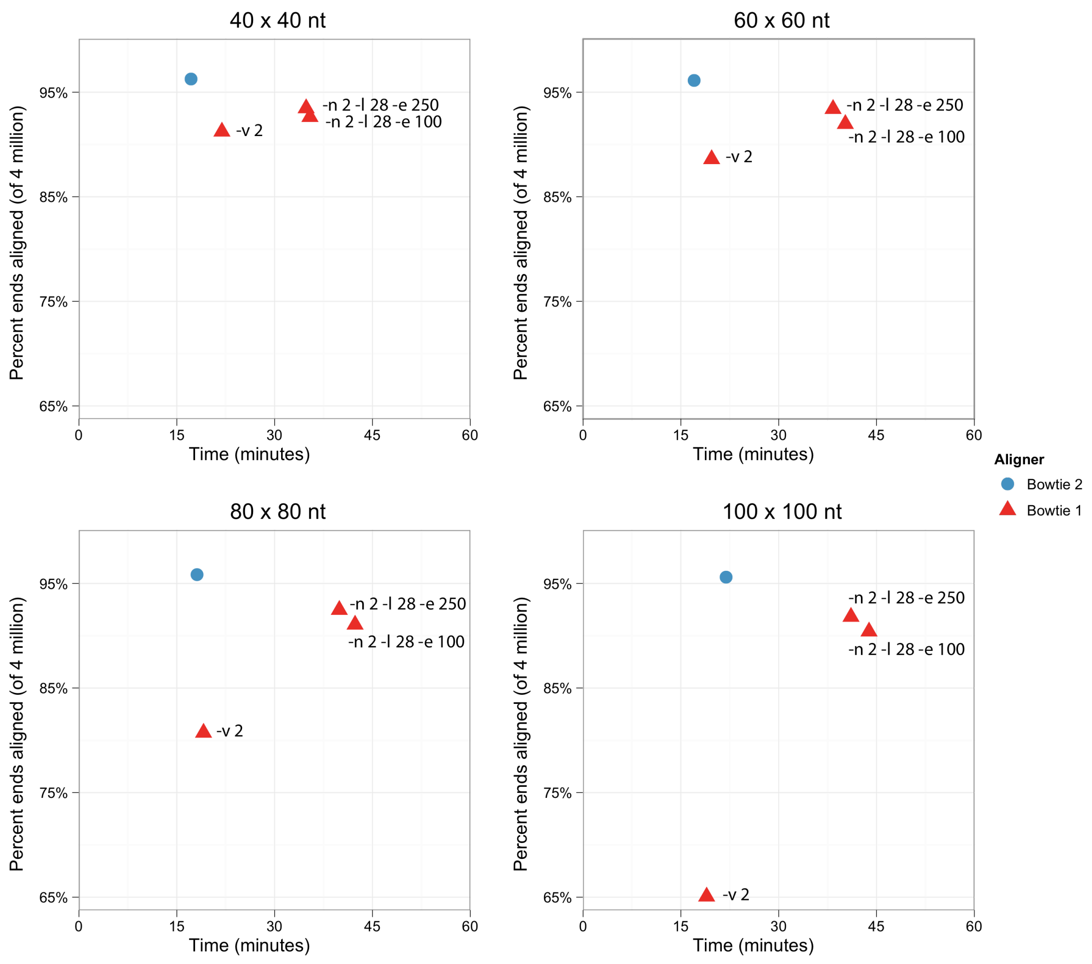

Figure 23: Comparison of Bowtie 1 and Bowtie 2 on paired-end reads.

Aligner	Options	Running time	% reads aligned (out of 2 million)	Peak virtual memory footprint (gigabytes)
Length: 40 x 40 nt				
Bowtie 2	--sensitive -I 0 -X 500	17m:11s	96.26%	3.34
Bowtie 1	-v 2 -M 1 --best -I 0 -X 500	21m:55s	91.23%	3.01
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best -I 0 -X 500	35m:25s	92.63%	3.01
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best -I 0 -X 500	34m:50s	93.46%	3.01
Length: 60 x 60 nt				
Bowtie 2	--sensitive -I 0 -X 500	17m:02s	96.12%	3.28
Bowtie 1	-v 2 -M 1 --best -I 0 -X 500	19m:43s	88.60%	3.01
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best -I 0 -X 500	40m:13s	91.96%	3.01
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best -I 0 -X 500	38m:20s	93.40%	3.01
Length: 80 x 80 nt				
Bowtie 2	--sensitive -I 0 -X 500	18m:06s	95.84%	3.26
Bowtie 1	-v 2 -M 1 --best -I 0 -X 500	19m:06s	80.72%	3.01
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best -I 0 -X 500	42m:20s	91.06%	3.01
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best -I 0 -X 500	39m:55s	92.47%	3.01
Length: 100 x 100 nt				
Bowtie 2	--sensitive -I 0 -X 500	21m:56s	95.60%	3.26
Bowtie 1	-v 2 -M 1 --best -I 0 -X 500	18m:57s	65.06%	3.01
Bowtie 1	-l 28 -n 2 -e 100 -M 1 --best -I 0 -X 500	43m:51s	90.40%	3.01
Bowtie 1	-l 28 -n 2 -e 250 -M 1 --best -I 0 -X 500	41m:05s	91.80%	3.01

Table 8: Comparison of Bowtie 1 and Bowtie 2 on paired-end reads: full results.

Comparison to additional tools

To assess the fraction of reads aligned by Bowtie 2 versus other tools, Bowtie 2, BWA [34], SOAP2 [37], GSNAP [90], MOSAIK [91], and SHRiMP 2 [92] were used to align a subset of 200,000 reads from the unpaired HiSeq 2000 dataset examined in Figure 20. The results are presented in Table 9. Default options for all tools except MOSAIK, where the recommended options for reads of around 100 nt were used. Not all tools are being run in comparable reporting modes; e.g. Bowtie 2, BWA and SOAP2 report one representative alignment for each input read by default, but GSNAP, SHRiMP, and MOSAIK report many alignments per read by default. Also, a substantial fraction of running time for MOSAIK and SHRiMP 2 is spent building the reference index, a cost that can be amortized in practice by aligning large collections of reads at once. For these reasons, and because only one set of parameters is tried for each tool, these results should not be considered to be a comprehensive comparison of these tools. But the results do convey a rough impression of how the tools compare in terms speed and fraction of reads aligned.

This experiment was run on a single Intel Xeon X5550 Nehalem 2.66GHz processor of a High-Memory Quadruple Extra Large Instance (m2.4xlarge) rented from Amazon's Elastic Compute Cloud (EC2) service [93]. The instance had 68.4 gigabytes of physical memory and was running the Basic 64-bit Amazon Linux AMI 2011.02.1 Beta.

Aligner	**Options**	**Running time**	**% reads aligned (out of 200,000)**	**Peak virtual memory footprint (gigabytes)**
Bowtie 2	(defaults)	39s	95.89%	3.24
BWA	(defaults)	1m:42s	91.81%	2.32
SOAP2	(defaults)	31s	84.45%	5.32
GSNAP	(defaults)	20m:56s	93.99%	4.91
MOSAIK	-mm 15 -act 35 -bw 35 -mhp 100	30m:27s	95.64%	61.70
SHRiMP2	(defaults)	251m:38s	97.67%	36.90

Table 9: Comparison of Bowtie 2 with various other tools.

Conclusions

Full-text genome indexing using the FM Index, pioneered in Bowtie, is now an increasingly common tool for aligning sequencing reads. Extending this method to perform sensitive gapped alignment and alignment of long reads without incurring serious computational penalties is a major technical challenge. Bowtie 2 is a new method that combines the advantages of the FM Index and SIMD dynamic programming, achieving very fast and memory-efficient gapped alignment of sequencing reads. Bowtie 2 improves on the previous Bowtie method in terms of speed and fraction of reads aligned (Figure 22 and Figure 23), beats other FM-Index-based tools on speed, sensitivity and accuracy (Figure 20 and Figure 21), and is significantly faster than non-FM-Index-based approaches while aligning a comparable fraction of reads (Table 9). Bowtie 2 is free, open source software available from the Bowtie 2 website at http://bowtie-bio.sf.net/bowtie2.

Chapter 4: Crossbow, a "big data" approach to alignment and variant calling

This chapter motivates the need for a parallel, "big data" approach to alignment and variant calling, which are common tasks in the field of comparative genomics. This chapter also describes the Crossbow software tool, which exemplifies how comparative genomics pipelines can be adapted to run in an efficient and scalable fashion atop the MapReduce programming framework and on commercial cloud computing services such as Amazon Web Services. I wrote the Crossbow software in collaboration with Michael Schatz, and a manuscript describing this work was published in 2010 [94].

A "big data" approach to comparative genomics

Modern sequencing datasets are very large and, thanks to improvements in sequencing throughput and cost, only getting larger. The pilot phase of the 1000 Genomes Project, for instance, generated about 4.9 trillion total nucleotides of data [60]. With larger datasets comes a new set of software design issues. The most obvious issue is the need for parallel software. Because computers are not getting faster at nearly the same rate as sequencers, software that runs on a single computer or a fixed number of computers will be unable to keep up. Instead, software must be made to run over many processors at once. Software must also tolerate severe hardware and software failures; when software runs on many computers over long periods of time, it can be unacceptably costly to have to abandon an analysis midstream because of the failure of a single computer. Finally, there is a need for computations to run under relatively stringent hardware constraints. Computations that make economical use of memory and processing power can be run in more contexts and are less likely to “starve” each other when running together on a single computer.

Crossbow represents an important step toward moving common comparative genomics analysis pipelines into the “big data” era. Crossbow is a software tool built atop the Apache Hadoop [24, 95] implementation of the MapReduce [27] programming model. MapReduce and Hadoop were designed to tackle problems caused by very large datasets, including the need for parallelism, fault tolerance, and economical use of resources. Hadoop facilitates parallel programming by forcing programmers to adhere to a simple but constrained programming model, described further below. Hadoop provides fault tolerance by storing permanent files and intermediate results in a redundant, distributed file system called the Hadoop Distributed File System (HDFS). And while Hadoop does not explicitly force programmers to write economical software, the MapReduce programming model encourages economical use of memory and processing power.

Crossbow implements a pipeline for identifying Single Nucloetide Polymorphisms (“SNP”), also known as Single Nucleotide Variants or SNVs. The pipeline’s goal is to (a) align reads to a reference genome, then (b) analyze the aligned reads to determine instances where the sequenced genome differs from the reference genome sequence by one nucleotide. For instance, there might be a position where the reference genome sequence has a C but the sequenced genome appears to have an A, as in Figure 24. SNPs are a commonly studied type of DNA variant with potentially significant implications. Some Mendelian diseases such as Sickle cell anemia, Systic fibrosis, and Tay-Sachs disease, are caused by SNP mutations.

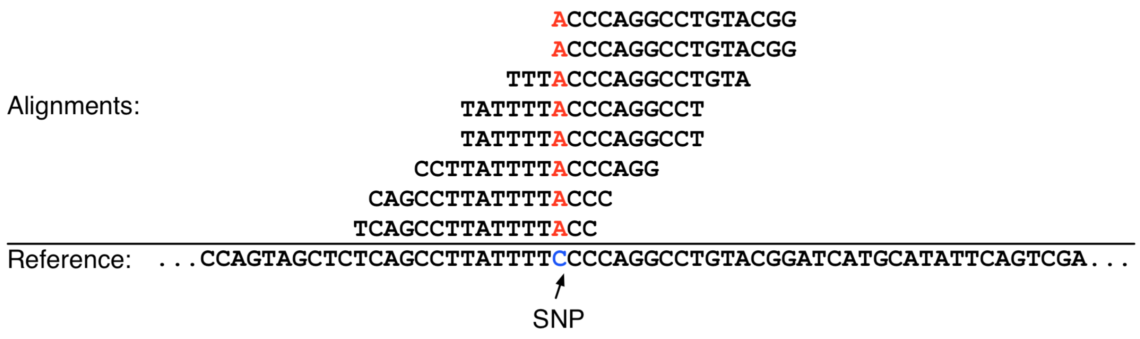

Figure 24: Illustration of a “C to A” SNP

The Crossbow SNP calling pipeline combines Bowtie’s [18] speed and memory-efficiency with the accuracy of the SNP caller SOAPsnp [96] to perform alignment and SNP calling for multiple whole-human datasets per day. In our experiments, for instance, Crossbow was able to align and call SNPs from 38-fold coverage[2] of a Han Chinese male genome [65] in as little as 3 hours (4 hours 30 minutes including transfer time) using a 320-core cluster rented from the Amazon Web Services [25] commercial cloud computing service. Crossbow can be run on any cluster with appropriate versions of Hadoop, Bowtie, and SOAPsnp installed. Crossbow is distributed with scripts that allow it to run on a single computer (exploiting multiple processors and processor cores where possible), on a Hadoop cluster, or on a cluster rented from Amazon’s Elastic MapReduce [97] service.

[2] The “coverage” is the average depth, i.e., the average number of reads covering a given position in the genome.

MapReduce

Broadly speaking, the MapReduce programming model requires that a program be expressed as an alternating series of “Compute” and “Aggregate” steps[3]. Each Compute step operates on a bin of “tuples,” where a tuple is simply an ordered collection of data with one or more “key” fields. A Compute step can only examine one bin at a time; it cannot combine data across bins or load multiple bins into memory at once. This is how MapReduce encourages parallelism; imposing these constraints on the programmer maximizes MapReduce’s freedom to schedule the per-bin computations to run simultaneously in parallel. The output from a Compute step is a stream of zero or more tuples.

An “Aggregate” step takes tuples emitted from one or more previous Compute steps and bins and sorts them a according to primary and secondary keys respectively. All binning and sorting is performed in a parallel, distributed fashion, and the Hadoop MapReduce implementation is quite efficient for this [98]. The programmer need not be concerned with the mechanics of binning and sorting. As long as the programmer properly specifies which fields contain primary and secondary keys, the MapReduce software handles the rest. The output from an Aggregate step is a collection of sorted bins of data, which then become the input for the following Compute step.

Steps in the Crossbow pipeline

The chief insight behind Crossbow is that a pipeline consisting of alignment followed by SNP calling can be framed as a short series of parallel Compute and

[3] “Compute” steps actually consist of Reduce and Map steps, but the distinction between Map and Reduce is unimportant here. MapReduce uses the name “Shuffle/Sort” for name the “Aggregate” step.

Aggregate steps. The first Compute step consists of running the Bowtie read aligner. In this phase, input tuples represent reads and output tuples represent alignments. This step is “parallel by read,” that is, if we have read A and read B, we can safely align A and B simultaneously on two different processors. Put another way: the result of aligning read A does not depend on the result of aligning read B, or vice versa.

Bowtie is run via Hadoop’s “streaming” mode, whereby any command line program or script can play the role of a Compute step, and the Hadoop infrastructure exchanges input and output tuples with the program via the “standard in” and “standard out” filehandles. Bowtie’s relatively small genome index is an asset here, since this makes it easier for Hadoop to run multiple Bowtie processes on the same computer. See below for further details on how Bowtie adapted to run in Hadoop’s streaming mode using modest hardware resources.

In the Aggregate step following the alignment step, alignments are binned according to the genomic region (“partition”) aligned to. Partitions are simply non-overlapping windows of the reference genome. Alignments spanning more than one partition are copied into both spanned partitions, such that each bin receives all the evidence that might have a bearing on the SNP calls made in the partition. The Aggregate step also sorts alignments along the forward strand of the reference genome in preparation for SNP calling.

The final Compute phase takes a sorted bin as input. As mentioned, the alignments in each bin are sorted along the forward strand of the reference genome; for instance, an alignment whose leftmost position is at genome offset 100 will appear before an alignment whose leftmost position is at genome offset 101. A sorted list of alignments like this is sometimes called a “pileup” and it is a useful way to represent alignment data because many subsequent analyses, including SNP calling, can proceed by walking along the genome from position to position and examining evidence from all the alignments overlapping that position. By sorting the alignments, we ensure that alignments overlapping a given position are close to each other in the list. See Figure 25 for an example bin.

For each bin, the Compute step runs the SOAPsnp [96] SNP caller on the bin’s alignments. Output from this step is a collection of tuples corresponding to all the positions where the subject genome nucleotide differed from the reference genome nucleotide.

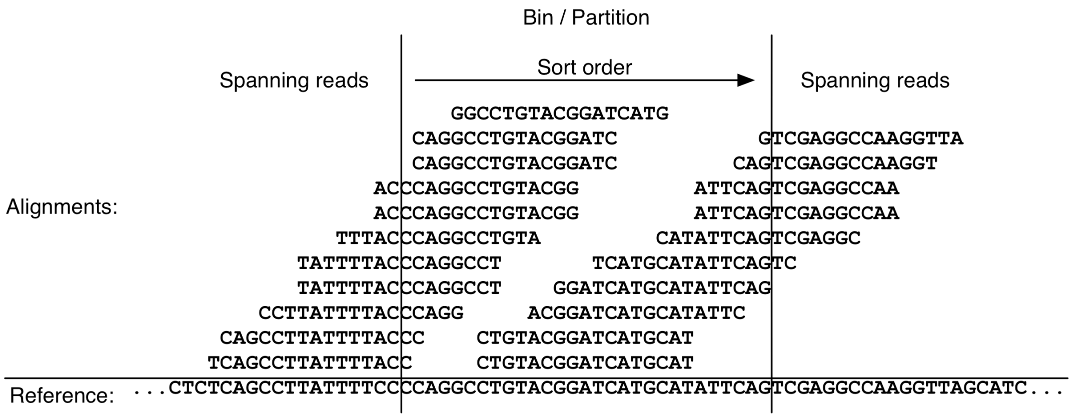

Figure 25: A sorted Crossbow bin ready for SNP calling

Modifications to existing software

Several new features were added to Bowtie to allow it to operate more smoothly as a Hadoop Compute step. First, Bowtie was extended to optionally use “memory-mapped files” or “shared memory” when loading the reference index into memory. This allows many independent Bowtie processes to run in parallel on a single multi-processor computer while sharing a single in-memory image of the reference index. This is much more memory-efficient than storing a separate copy of the index per Bowtie process. Also, modifications were made to allow Bowtie and SOAPsnp to interoperate with Hadoop (via standard in and standard out) and with each other (via properly formatted tuples).

Cloud support

Crossbow is distributed with scripts that allow it to run on a single multi-processor computer, on a Hadoop cluster, or on a cluster rented from Amazon's Elastic MapReduce [97] service. When running Crossbow on Amazon's Elastic MapReduce service, only the script for launching the job is run locally; all other computation is executed remotely on computers rented from Amazon's Elastic Compute Cloud [93] service. See Figure 26 for an illustration of how Crossbow runs using cloud resources.

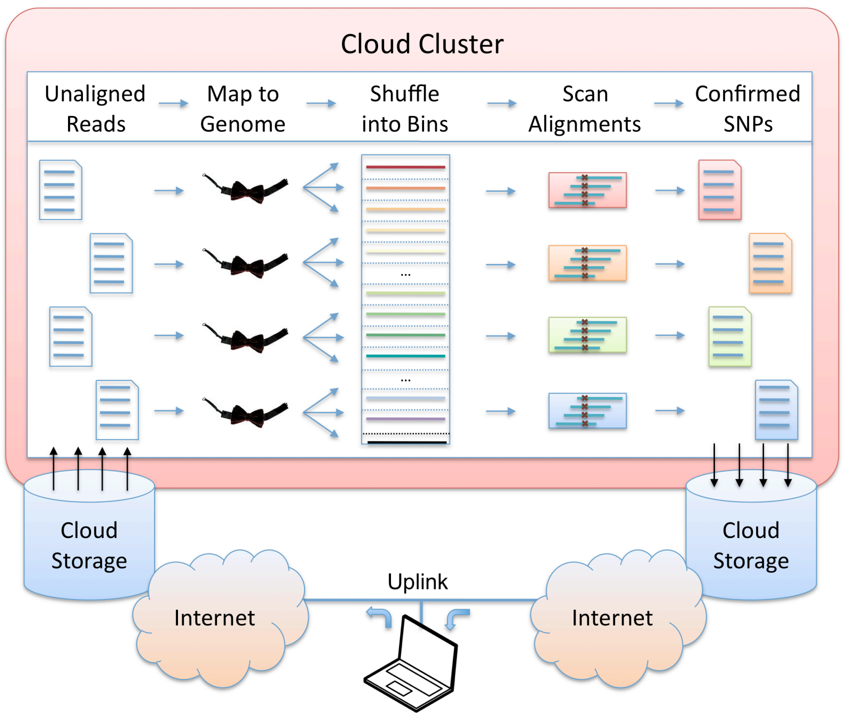

Figure 26: Illustration of how Crossbow is run using cloud resources.

<u>Performance results using simulated data</u>

Real sequencing datasets are generated by sequencing instruments and derived from biological samples; “simulated” sequencing datasets, on the other hand, are generated *in silico* by software, and are derived directly from a reference genome sequence. Simulated reads might also be modified to reflect synthetic variants between individuals and sequencing errors. Though simulated data lack some of the key properties possessed by real data, simulations allow us to perform controlled experiments where certain outcomes, such as the set of “true” SNPs, are known beforehand.

To measure Crossbow's accuracy with respect to a known set of "true" SNPs, we conducted two experiments using simulated paired-end read data from human chromosomes 22 and X. Results are shown in Table 10. For both experiments, 40-fold coverage of 35 nt paired-end reads were simulated from the human reference sequence (NCBI build 36.3). Quality values and insert lengths were simulated based on empirically observed qualities and inserts in the Wang *et al.* dataset [65].

Experimental Parameters						
Reference chromosome	**Chromosome 22**			**Chromosome X**		
Reference base pairs	49.7 million			155 million		
Chromosome copy number	Diploid			Haploid		
HapMap SNPs introduced	36,096			71,976		
Heterozygous	24,761			0		
Homozygous	11,335			71,976		
Novel SNPs introduced	10,490			30,243		
Heterozygous	6,967			0		
Homozygous	3,523			30,243		
Simulated coverage	40-fold			40-fold		
Read type	35 nt paired			35 nt paired		
SNP Calling	True # sites	Crossbow sensitivity	Crossbow precision	True # sites	Crossbow sensitivity	Crossbow precision
All SNP sites	46,586	99.0%	99.1%	102,219	99.0%	99.6%
only HapMap SNP sites	36,096	99.8%	99.9%	71,976	99.9%	99.9%
only novel SNP sites	10,490	96.3%	96.3%	30,243	96.8%	98.8%
only homozygous	14,858	98.7%	99.9%	N/A	N/A	N/A
only heterozygous	31,728	99.2%	98.8%	N/A	N/A	N/A
only novel het	6,967	96.6%	94.6%	N/A	N/A	N/A
all other	39,619	99.4%	99.9%	N/A	N/A	N/A

Table 10: Experimental parameters for Crossbow experiments using simulated reads from human chromosomes 22 and X.

SOAPsnp can exploit user-supplied information about known SNP loci and allele frequencies to refine its prior probabilities and improve accuracy. Therefore, the read simulator was designed to simulate both known HapMap [99] SNPs and novel SNPs. This mimics resequencing experiments where many SNPs are known but some are novel. Known SNPs were selected at random from actual HapMap alleles for human chromosomes 22 and X. Positions and allele frequencies for known SNPs were calculated according to the same HapMap SNP data used to simulate SNPs.

For these simulated data, Crossbow agrees substantially with the true calls, with greater than 99% precision and sensitivity overall for chromosome 22. Performance for HapMap SNPs is noticeably better than for novel SNPs, owing to SOAPsnp's ability to adjust SNP-calling priors according to known allele frequencies. Performance is similar for homozygous and heterozygous SNPs overall, but novel heterozygous SNPs yielded the worst performance of any other subset studied, with 96.6% sensitivity and 94.6% specificity on chromosome 22. This is as expected, since novel SNPs do not benefit from prior knowledge, and heterozygous SNPs are more difficult than homozygous SNPs to distinguish from the background of sequencing errors.

Results using whole-genome human resequencing data

To demonstrate performance on real-world data, we used Crossbow to align and call SNPs from the set of 2.7 billion reads and paired-end reads sequenced from a Han Chinese male by Wang *et al.* [65]. Previous work demonstrated that SNPs called from this dataset using a combination of SOAP [36] and SOAPsnp [96] are highly concordant with genotypes called by an Illumina 1 M BeadChip genotyping assay of the same individual [96]. Since Crossbow uses SOAPsnp as its SNP caller, we expected Crossbow to yield very similar, but not identical, output. Differences may occur because: Crossbow uses Bowtie whereas the previous study used SOAP to align the reads; the Crossbow version of SOAPsnp has been modified somewhat to operate within a MapReduce context; in this study, alignments are binned into non-overlapping 2 million nt partitions rather than into chromosomes prior to being given to SOAPsnp; and the SOAPsnp study used additional filters to remove some additional low confidence SNPs. Despite these differences, Crossbow achieves comparable agreement with the BeadChip assay.

We downloaded 2.66 billion reads from a mirror of the YanHuang website [100]. These reads cover the assembled human genome sequence to 38-fold coverage. They consist of 2.02 billion unpaired reads with sizes ranging from 25 to 44 nt, and 658 million paired-end reads. The most common unpaired read lengths are 35 and 40 nt, comprising 73.0% and 17.4% of unpaired reads, respectively. The most common paired-end read length is 35 nt, comprising 88.8% of all paired-end reads. The distribution of paired-end separation distances is bimodal with peaks in the 120 to 150 nt and 420 to 460 nt ranges.

Table 11 shows a comparison of SNPs called by either of the sequencing-based assays – Crossbow labeled "CB" and SOAP+SOAPsnp labeled "SS" – against SNPs obtained with the Illumina 1 M BeadChip assay from the SOAPsnp study [96]. The "sites covered" column reports the proportion of BeadChip sites covered by a sufficient number of sequencing reads. Sufficient coverage is roughly four reads for diploid chromosomes and two reads for haploid chromosomes. The "Agreed" column shows the proportion of covered BeadChip sites where the BeadChip call equaled the SOAPsnp or Crossbow call. The "Missed allele" column shows the proportion of covered sites where SOAPsnp or Crossbow called a position as homozygous for one of two heterozygous alleles called by BeadChip at that position. The "Other disagreement" column shows the proportion of covered sites where the BeadChip call differed from the SOAPsnp/Crossbow in any other way. Definitions of the "Missed allele" and "Other disagreement" columns correspond to the definitions of "false negatives" and "false positives," respectively, in the SOAPsnp study.

Illumina 1M genotype	Sites	Sites covered (SS)	Sites covered (CB)	Agreed (SS)	Agreed (CB)	Missed allele (SS)	Other disagree-ment (SS)	Missed allele (CB)	Other disagree-ment (CB)
Chr X									
HOM reference	27,196	98.65%	99.83%	99.99%	99.99%	N/A	.004%	N/A	.011%
HOM mutant	10,737	98.49%	99.19%	99.89%	99.85%	N/A	.113%	N/A	.150%
Total	37,933	98.61%	99.65%	99.97%	99.95%	N/A	.035%	N/A	.050%
Autosomal									
HOM reference	540,878	99.11%	99.88%	99.96%	99.92%	N/A	.044%	N/A	.078%
HOM mutant	208,436	98.79%	99.28%	99.81%	99.70%	N/A	.194%	N/A	.296%
HET	250,667	94.81%	99.64%	99.61%	99.75%	.374%	.017%	.236%	.014%
Total	999,981	97.97%	99.70%	99.84%	99.83%	.091%	.069%	.059%	.108%

Table 11: Comparison of Crossbow and SOAP/SOAPsnp to Illumina 1M genotyping assay.

Both Crossbow and SOAP+SOAPsnp exhibit a very high level of agreement with the BeadChip genotype calls. The small differences in number of covered sites (<2% higher for Crossbow) and in percentage agreement (<0.1% lower for Crossbow) are likely due to the SOAPsnp study's use of additional filters to remove some SNPs prior to the agreement calculation, and to differences in alignment policies between SOAP and Bowtie. After filtering, Crossbow reports a total of 3,738,786 SNPs across all autosomal chromosomes and chromosome X, whereas the SNP GFF file available from the YanHaung site [100] reports a total of 3,072,564

SNPs across those chromosomes. This difference is also likely due to the SOAPsnp study's more stringent filtering.

Crossbow performance using cloud clusters

The above results were computed on a Hadoop 0.20 cluster with 10 worker nodes located in our laboratory, where it required about 1 day of wall clock time to run. Each node is a four-core 3.2 GHz Intel Xeon (40 cores total) running 64-bit Redhat Enterprise Linux Server 5.3 with 4 gigabytes of physical memory and 366 gigabytes of local storage available for the Hadoop Distributed Filesystem (HDFS). We also performed this computation using Amazon's EC2 [93] service on clusters of 10, 20 and 40 nodes (80, 160, and 320 cores) running Hadoop 0.20. In each case, the Crossbow pipeline was executed end-to-end using scripts distributed with the Crossbow package. In the 10-, 20- and 40-node experiments, each individual node was an EC2 Extra Large High CPU Instance, that is, a virtualized 64-bit computer with 7 gigabytes of memory and the equivalent of 8 processor cores clocked at approximately 2.5 to 2.8 Ghz. At the time of this writing, the cost of such nodes was $0.68 per node per hour.

Before running Crossbow, the read data must be stored on a filesystem the Hadoop cluster can access. When the Hadoop cluster is rented from Amazon's EC2 service, users will typically upload input data to Amazon's Simple Storage Service (S3) [101], a service for storing large datasets over the Internet. For small datasets, data transfers are typically quick, but for large datasets (for example, more than 100 gigabytes of compressed read data), transfer time can be significant. An efficient method to copy large datasets to S3 is to first allocate an EC2 cluster of many nodes and have each node transfer a subset of the data from the source to S3 in parallel. Crossbow is distributed with a Hadoop program and driver scripts for performing these bulk parallel copies while also preprocessing the reads into the form required by Crossbow. We used this software to copy 103 gigabytes of compressed read data from a public FTP server located at the European Bioinformatics Institute in the UK to an S3 repository located in the US in about 1 hour 15 minutes (an effective transfer rate of about 187 megabits per second). The transfer cost approximately $28: about $3.50 in cluster rental fees and about $24 in transfer fees.

Transfer time depends heavily on both the size of the data and the speed of the Internet uplink at the source. Public archives like NCBI and the European Bioinformatics Institute have very high-bandwidth uplinks to network backbones, as do many academic institutions. However, even at these institutions, the bandwidth available for a given server or workstation can be considerably less (commonly 100 megabits per second or less). Delays due to slow uplinks can be mitigated by transferring large datasets in stages as reads are generated by the sequencer, rather than all at once.

To measure how the whole-genome Crossbow computation scales, separate experiments were performed using 10, 20 and 40 EC2 Extra Large High CPU nodes. Table 12 presents the wall clock running time and approximate cost for each experiment. The results show that Crossbow is capable of calling SNPs from 38-fold coverage of the human genome in under 3 hours of wall clock time and for about $85.

Whole Genome Genotyping Runtime and Costs

EC2 Nodes	1 master, 10	1 master, 20	1 master, 40

	workers	workers	workers
Worker CPU cores	80	160	320
Wall clock time	6h:30m	4h:33m	2h:53m
Approx. cluster setup time	18m	18m	21m
Approx. Crossbow time	6h:12m	4h:15m	2h:32m
Approximate cost (US/Europe)	\$52.36/ \$60.06	\$71.40/\$81.90	\$83.64/\$95.94

Table 12: Timing and cost for Crossbow experiments using reads from the Wang *et al.* study.

Figure 27 illustrates scalability of the computation as a function of the number of processor cores allocated. Units on the vertical axis are the reciprocal of the wall clock time. Whereas wall clock time measures elapsed time, its reciprocal measures throughput, i.e., experiments per hour. The straight diagonal line extending from the 80-core point represents hypothetical linear speedup, that is, extrapolated throughput under the assumption that doubling the number of processors also doubles throughput. In practice, parallel algorithms usually exhibit worse-than-linear speedup because portions of the computation are not fully parallel. In the case of Crossbow, deviation from linear speedup is primarily due to load imbalance among processors in the map and reduce phases, which can cause a handful of work-intensive "straggler" tasks to delay progress. The reduce phase can also experience imbalance due to, for example, variation in coverage. However, the fact that Crossbow achieves appreciable speedup even in the 100s-of-processors range is a positive result.

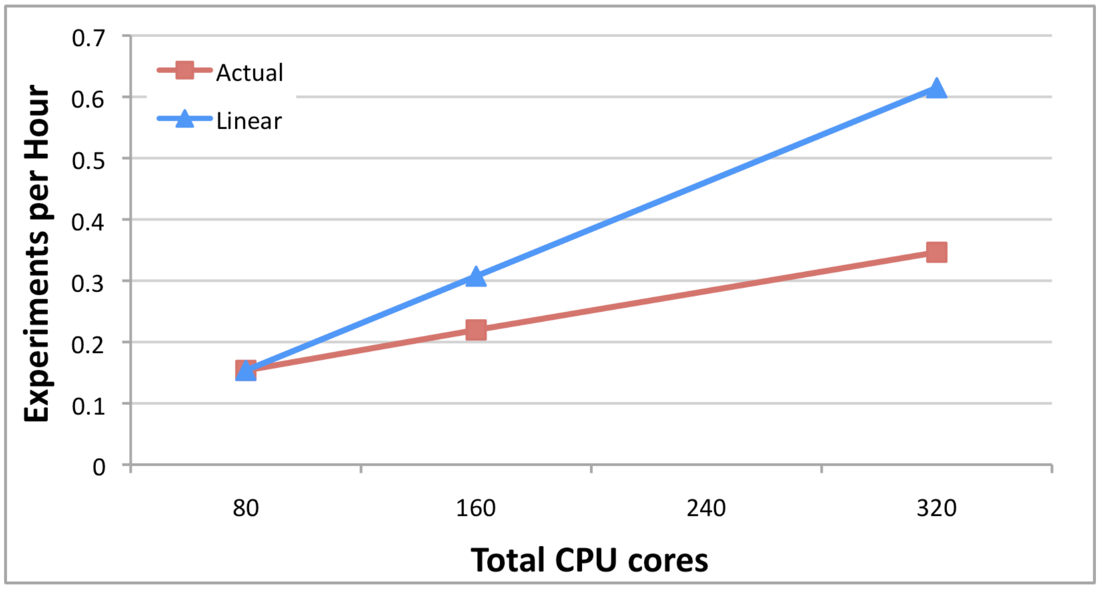

Figure 27: Crossbow speedup versus number of processor cores allocated.

Conclusions

It is increasingly critical to design software that can deal with all the implications of “big data.” Crossbow represents an important initial step in this direction. It is the first end-to-end comparative genomics pipeline implemented using the MapReduce programming model, and one of the first examples of a genomics tools designed to run on a commercial cloud computing service. The relative simplicity of Crossbow’s design is an indication that similar comparative genomics analysis pipelines could also be adapted to run using the MapReduce programming model. Also, the fact that Crossbow’s achieves appreciable speedup even in the 100s-of-processors range is an indication that other pipelines can make profitable use of 100s of processors. Crossbow is free, open source software available from the Crossbow website at http://bowtie-bio.sf.net/crossbow.

Chapter 5: Myrna: a "big data" approach to differential gene expression

This chapter motivates a "big data" approach to messenger RNA sequencing data analysis, which is perhaps the single most commonly performed task in the field of comparative genomics. I will also describe Myrna, a software pipeline for detecting differential gene expression that runs in an efficient and scalable fashion atop the MapReduce programming framework and on commercial cloud computing services such as Amazon Web Services. I am the primary author of the Myrna software. Jeffrey Leek and Kasper Hansen developed and helped with the implementation of the statistical approaches, which are not discussed in depth here. A manuscript describing this work, including the statistical aspects, was published in 2010 [69].

RNA-seq

Sequencing of messenger RNAs (often abbreviated "mRNA-seq" or "RNA-seq") yields reads derived from messenger RNA molecules ("mRNAs") present in one or more biological samples. An mRNA is a "transcribed" copy of a gene in the genome. The mRNA molecule (or "transcript") is an intermediate product between a gene and the protein that the gene encodes. Each time we observe a particular mRNA in a sample, this constitutes evidence that the gene from which the mRNA was transcribed is "expressed" in the sample. Thus, whereas DNA sequencing is concerned with what DNA variants are present in the genome, RNA sequencing is concerned with which genes are turned on or off, and to what degree.

Subsequent discussion in this chapter will refer to "differential gene expression." For the purpose of this discussion, we will assume that the differential expression analysis takes place at the granularity of genes. This is not always the unit of interest; some projects may be more interested in differential expression of exons (smaller portions of genes that combine to form transcripts) or of transcripts themselves (which have a many-to-one relationship with genes). Much of the discussion in this chapter apply equally well to exons and genes. For discussion of trasnscript-level analysis, see the Cufflinks publication [68].

Variability in RNA-seq

The degree to which genes are expressed in a sample (i.e. its "gene expression profile") is partly a property of the cell type being studied. In the human body, for example, a liver cell has one gene expression profile and a blood cell has a distinct gene expression profile. A critical question that emerges in many life science projects is: given two groups of samples, which genes are "differentially expressed" between the two? That is, which genes are turned on in one group and off in the other? Answering such questions allows us to better understand the relationship between gene expression and other phenomena of interest, such as disease.

When studying gene expression, variability is a key consideration. Say that we are interested in studying differences in gene expression between livers and brains. We are encumbered by the fact that (a) given the same inputs, not every run of the sequencing machine produces the exact same results, and (b) not all livers are exactly alike and not all brains are exactly alike. Put another way, we have to contend with (a) technical variability and (b) biological variability.

If we take the same biological sample and use RNA-seq to sequence it twice, we will get somewhat different gene expression measurements. This is owing to “technical variability” introduced by the sequencer. Statistical techniques can be used to characterize and remove technical variability provided that we have “technical replicates” – i.e. many sequencing datasets using separate runs of the sequencer.

Biological variability consists of all the “natural” variability between individuals. For instance, if we use RNA-seq to sequence the livers of individuals A and B and the spleens of individuals A and B, some of the observed gene expression differences between livers and spleens will be due to “liverness” versus “spleenness,” but some will be due to other apects of “individual A-ness” versus “individual B-ness.” We can use statistical techniques to distinguish between the two as long as we have “biological replicates” – i.e. many sequencing datasets derived from different individuals but involving the same groups of interest (liver and spleen in this case). Taking biological variability into account is a critical need in modern comparative genomics research [102].

The need to take technical and biological variability into account necessitates sequencing of many replicates, which in turn yields larger datasets. It is interesting to note that, in this case, dataset size is driven not just by the capabilities of the sequencer, but is also by the need for accuracy and the difficulty of the scientific question being posed.

Design of Myrna

RNA-seq yields a collection of sequence reads per input sample. Each read is derived from mRNA molecule, which in turn is derived from a gene in the genome.

For differential expression, samples are divided into two or more “groups,” e.g. “liver” and “spleen.” To account for variability, there are generally many samples per group.

Given these reads, differential-expression analysis often proceeds in three stages. First, reads are computationally categorized according to the gene from which each likely originated. This categorization step could be conducted comparatively with respect to a reference genome [103, 104], via *de novo* assembly [105, 106], or via a combination of both [67, 68, 107]. After reads are categorized, a normalized count of the number of reads assigned to each gene is calculated. This count is a proxy for the gene’s abundance in the sample. The count is also normalized in order to eliminate effects due to technical variability. Finally, a statistical test is applied with respect to each gene in order to determine whether the gene is differentially expressed between groups.

Myrna is a “big data”-ready software pipeline for differential expression analysis of RNA-seq datasets. Myrna can be run in one of three modes: “Cloud mode” using Amazon Elastic MapReduce; “Hadoop mode” using a Hadoop cluster; or “Singleton mode” using a single computer. Cloud mode does not require any special software installation; the appropriate software is either pre-installed or automatically installed on the rented Amazon EC2 computers before Myrna is run. Singleton mode is also parallelized and can exploit a user-specified number of processors. Myrna is designed with the Apache Hadoop [95] open source implementation of the MapReduce [27] programming model in mind.

Steps in the Myrna pipeline

Myrna's pipeline is depicted in Figure 28. Each stage exploits a different type of parallelism with the aim of maximizing scalability. The first stage ("Preprocess") takes a list of files containing input reads and deposits a "preprocessed" version of the input to a filesystem visible to the cluster. When preprocessed, reads are converted to a one-pair-per-line format and are annotated with metadata, including the read's sample ID and the name of the file where it originated. The sample name might be of the form "Normal-1-1," indicating that the sample is from the first technical replicate of the first biological replicate of the "Normal" group. The sample name "Cancer-3-2" would indicate that the sample is from the the second technical replicate of the third biological replicate of the "Cancer" group. The Preprocess stage is parallel across input files. That is, input files is downloaded and preprocessed simultaneously and in parallel with other input files wherever possible.

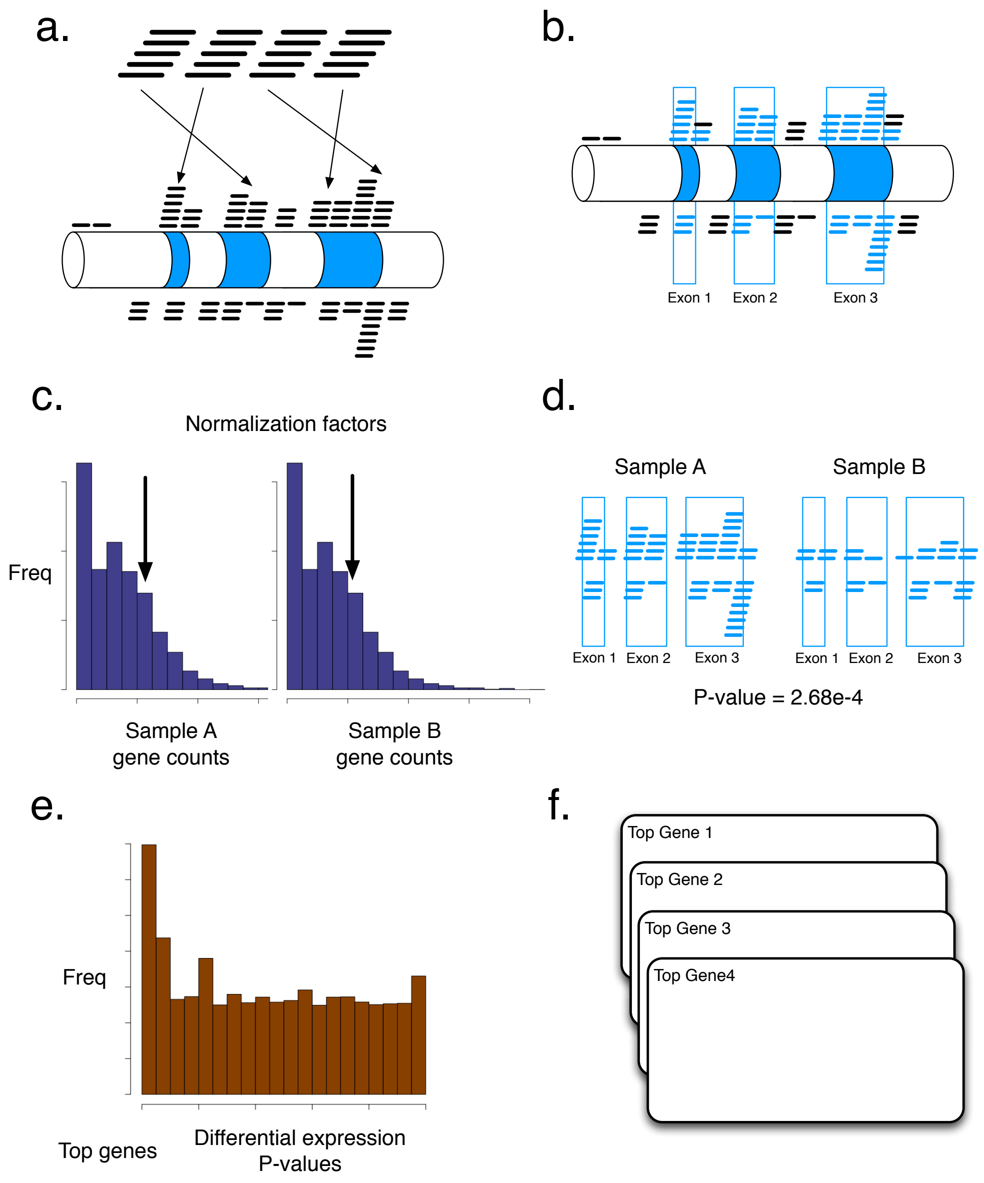

Figure 28: Illustration of the steps in the Myrna pipeline.

The second stage ("Align," Figure 28a) aligns reads to a reference genome using Bowtie [18]. As discussed previously, Bowtie employs a compact index of the reference genome and requires about a few gigabytes of memory to store an index of the human genome. The user may specify options to be passed to Bowtie in this stage; the default is "-m 1", which discards alignments for reads that align multiple places.

The alignment stage is parallel across reads; that is, reads are aligned simultaneously in parallel where possible.

The third stage (“Overlap,” Figure 28b) calculates overlaps between alignments from the Align stage and a pre-defined collection of “gene interval sets.” In each instance where the rightmost (3'-most) nucleotide of an alignment overlaps any nucleotide of a gene interval set, an overlap record associating the (labeled) alignment with the gene is output. A gene interval set is the minimal set of intervals such that all contained nucleotides are covered by all transcripts annotated for the gene. Intervals where two or more genes overlap are omitted from all gene interval sets. Thus, Myrna essentially implements the “union intersection” model proposed previously [103]. The Overlap stage is parallel across alignments; that is, overlaps for distinct alignments are calculated simultaneously and in parallel where possible.

The fourth stage (“Normalize,” Figure 28c) constructs a sorted vector of per-gene overlap counts for each sample. A normalization factor is then calculated for each sample, typically by extracting a quantile from, or otherwise summarizing, the sample-specific gene count distribution. By default, Myrna extracts the the 75th percentile of the distribution of non-zero gene counts, as suggested previously [103]. Alternatively, the user may specify that Myrna use a different quantile or value, such as the median or total, as the normalization factor. The Normalize stage is parallel across samples.

The fifth stage (“Statistics,” Figure 28d) examines counts for each gene and calculates and outputs a P-value describing the probability that differences in counts observed between groups are due to chance. The basic approach is to fit a generalized linear model relating the counts to the group from which the count was derived. Further details of the statistical approach, which is chiefly the work of colleagues Jeffrey Leek and Kasper Hansen, can be found in the Myrna publication [69]. The Statistics stage is parallel across genes.

The sixth stage ("Summarize," Figure 28e) examines a sorted list of all P-values generated in the Statistics stage and compiles a list of the top N genes ranked by false discovery rate, where the parameter N is set by the user. In addition to the global significance results, more detailed statistical results and figures are returned for the top N genes. Since there is not much parallelism inherent in this task, Myrna performs the Summarize stage serially (on a single processor). The lack of parallelism is mitigated by the fact that this stage typically does not take very long.

In the seventh stage ("Postprocess"), Myrna outputs a series of output files, including: (a) files listing all overlaps for each top gene, including alignment information that might indicate the presence of sequence variants, such as SNPs; (b) a table with estimated "Reads Per Kilobase of model per Million mapped reads" ("RPKM") values for each gene in the annotation; (c) a sorted table of all P-values for all genes, along with a histogram plot; and (d) a series of plots showing the coverage for each of the top N genes, broken down by replicate and by group. These results are compressed and stored in the user-specified output directory.

Using MapReduce for permutation testing

One notable aspect of the differential gene expression test in the Statistics step (Figure 28d) is that P-values can be assigned either parametrically or non-parametrically, depending on options specified by the user. When the non-parametric approach is used, Myrna performs a computationally intensive permutation procedure. Here, Myrna calculates a large number of "null statistics" by repeatedly shuffling the sample labels and re-calculating the test statistic. At the end of this process, for each gene, Myrna has calculated one "observed statistic" using the original sample labels and a potentially large collection of null statistics using shuffled labels. For each gene, Myrna then assigns a P-value by calculating the observed statistic's quantile with respect to the distribution of null statistics. I.e. if the gene's observed statistic is greater than half the null statistics, its P-value is 0.5. If the observed statistic is greater than 95% of the null statistics, its P-Value is 0.05.

The most computationally intensive aspect of this calculation is the construction of the null distribution, which boils down to sorting the list of all observed and null statistics. Hadoop's facilities for efficient, distributed binning and sorting come in handy here; specifically, we can rapidly situate all the observed statistics within the larger null distribution by simply sorting all the observed and null statistics in one large bin, then scanning the bin.

Cloud computing performance

We demonstrate Myrna's performance and scalability using a large population-based RNA-seq experiment [108]. This experiment sequenced 69 lymphoblastoid cell lines derived from unrelated Nigerian individuals studied by the HapMap project [99], the largest publicly available RNA-seq experiment at the time the Myrna manuscript was prepared. Each sample was sequenced at two separate labs (Argonne and Yale) on Illumina Genome Analyzer II instruments. For each sample, both labs contributed at least one lane of unpaired reads. In cases where a lab contributed more than one lane, we excluded data from all lanes beyond the first. The total input consisted of 1.1 billion reads; one sequencing center generated 35 nt unpaired reads and the other 46 nt unpaired reads. All reads were truncated to 35 nt prior to alignment. For each gene, a minimal set of genomic intervals was calculated such that all nucleotides covered by the interval set were covered by all annotated gene transcripts. Where intervals for two or more genes overlapped, the overlapping subinterval was excluded from all sets. The result is one non-overlapping interval set per gene encoding the portions of the gene that are "constitutive" (included in all transcripts) according to the annotation, and unique to that gene.

We ran the entire Myrna pipeline on this dataset using Amazon Elastic MapReduce [97] clusters of 10, 20 and 40 worker nodes (80, 160, and 320 cores). In each case, the Myrna pipeline was executed end-to-end using scripts distributed with the Myrna package. The nodes used were EC2 Extra Large High CPU Instances, that is, virtualized 64-bit computers with 7 GB of memory and the equivalent of 8 processor cores clocked at approximately 2.5 to 2.8 Ghz. At the time of this writing, the cost of such nodes was $0.68 per node per hour, with an Elastic MapReduce surcharge of $0.12 per node per hour.

Before running Myrna, the input read data must be stored on a file system accessible to the cluster. Users will typically upload and preprocess the input data to Amazon's Simple Storage Service (S3) [101] before running the rest of the Myrna pipeline. An efficient method to move data into S3 is to first allocate an Elastic MapReduce [97] cluster of many nodes and have each node transfer a subset of the data from the source to S3 in parallel. The first stage of the Myrna pipeline performs such a bulk copy while also preprocessing the reads into the form required by later stages of the Myrna pipeline. This software was used to copy 43 gigabytes of compressed read data from a public HTTP server located at the University of Chicago to an S3 repository located in the US in about 1 hour 15 minutes (approximately 82 megabits per second effective transfer rate). The transfer cost approximately $11: about $6.40 in cluster rental fees and about $4.30 in data transfer fees.

Transfer time depends heavily on both the size of the data and the speed of the Internet uplink at the source. Public archives like National Center for Biotechnology Information and the European Bioinformatics Institute as well as many universities have very high bandwidth uplinks to Internet backbones, making it efficient to copy data between those institutions and S3. However, depending on the uplink speed at the point of origin of the sequencing data, it may be more desirable to run Myrna on a local computer or cluster rather than on a commercial cloud service.

To measure scalability, separate experiments were performed using 10, 20 and 40 EC2 Extra Large High CPU worker nodes (plus one master node). Table 13 presents the wall clock running time and approximate cost for each experiment. The experiment was performed once for each cluster size. The results show that Myrna is capable of calculating differential expression from 1.1 billion RNA-seq reads in less than 2 hours of wall clock time for about $66. Figure 29 illustrates scalability as a function of the number of processor cores allocated. Units on the vertical axis are the reciprocal of the wall clock time. Whereas wall clock time measures elapsed hours per experiment, its reciprocal measures experiments per hour. The straight line extending from the 80-core point represents hypothetical linear speedup, extrapolated assuming that doubling the number of processors also doubles throughput. In practice, parallel algorithms usually exhibit worse-than-linear speedup because portions of the computation are not fully parallel. For Myrna, deviation from linear speedup is primarily due to load imbalance among processors in the Align stage, but also due to a deficit of parallelism in some downstream stages (for example, Normalize and Postprocess). However, as was the case with Crossbow, the fact that Myrna achieves appreciable speedup even in the 100s-of-processors range is a positive result.

Myrna Runtime, Cost for 1.1 billion reads from Pickrell *et al.* study

EC2 Nodes	1 master, 10 workers	1 master, 20 workers	1 master, 40 workers
Worker processor cores	80	160	320
Wall clock time	4h:20m	2h:32m	1h:38m
Cluster setup	4m	4m	3m
Align	2h:56m	1h:31m	54m
Overlap	52m	31m	16m
Normalize	6m	7m	6m
Statistics	9m	6m	6m
Summarize & Postprocess	13m	14m	13m
Approximate cost (depends on location)	\$44.00/\$49.50	\$50.40/\$56.70	\$65.60/\$73.80

Table 13: Timing and cost for Myrna experiments using reads from the Pickrell *et al.* study.

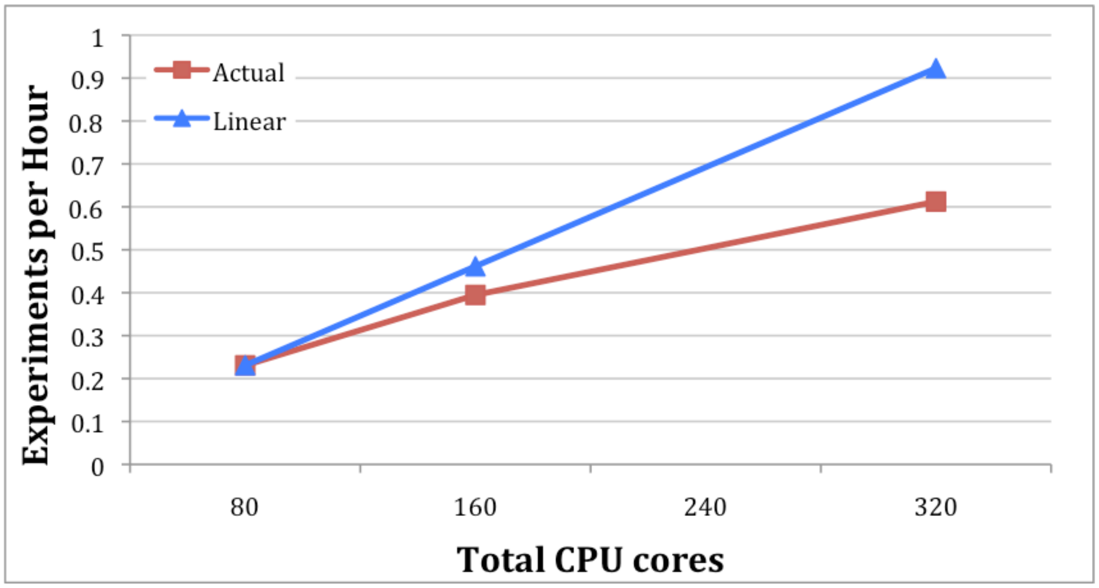

Figure 29: Myrna speedup versus number of processor cores allocated.

Conclusions

Myrna brings a very common comparative genomics analysis pipeline – RNA-seq differential gene expression analysis – into the era of data intensive computing by adapting it to work in the MapReduce framework. Myrna is notable because the pipeline is substantially more complex than Crossbow, for example. Like Crossbow, Myrna performs alignment, followed by a step ("Overlap") that is parallel be genome partition. After there, however, the pipelines diverge and Myrna goes on to perform computations parallel by sample ("Normalize") and by gene ("Statistics"). Myrna's use of Hadoop's facilities for distributed sorting to facilitate non-parametric significance testing is particularly notable. Despite its complexity, Myrna is still able to process a very large RNA-seq dataset in a few hours for less than $100. This indicates that more comparative genomics pipelines, including complex pipelines, can be profitably adapted to the MapReduce framework in the future.

Myrna is free, open source software available from the Myrna website at http://bowtie-bio.sf.net/myrna.

Bibliography

1. Metzker ML: **Sequencing technologies - the next generation**. *Nat Rev Genet* 2010, **11**(1):31-46.
2. Wang Z, Gerstein M, Snyder M: **RNA-Seq: a revolutionary tool for transcriptomics**. *Nat Rev Genet* 2009, **10**(1):57-63.
3. Hansen KD, Timp W, Bravo HC, Sabunciyan S, Langmead B, McDonald OG, Wen B, Wu H, Liu Y, Diep D *et al*: **Increased methylation variation in epigenetic domains across cancer types**. *Nat Genet* 2011, **43**(8):768-775.
4. Lister R, Pelizzola M, Dowen RH, Hawkins RD, Hon G, Tonti-Filippini J, Nery JR, Lee L, Ye Z, Ngo QM *et al*: **Human DNA methylomes at base resolution show widespread epigenomic differences**. *Nature* 2009, **462**(7271):315-322.
5. Park PJ: **ChIP-seq: advantages and challenges of a maturing technology**. *Nat Rev Genet* 2009, **10**(10):669-680.
6. Illumina I: **http://www.illumina.com/documents/systems/hiseq/datasheet_hiseq_systems.pdf**. Accessed January 7, 2012.
7. Venter JC, Adams MD, Myers EW, Li PW, Mural RJ, Sutton GG, Smith HO, Yandell M, Evans CA, Holt RA *et al*: **The sequence of the human genome**. *Science* 2001, **291**(5507):1304-1351.
8. Institute NHGR: **The Human Genome Project Completion: Frequently Asked Questions (http://www.genome.gov/11006943)**. Accessed January 7, 2012.
9. Miller JR, Koren S, Sutton G: **Assembly algorithms for next-generation sequencing data**. *Genomics* 2010, **95**(6):315-327.
10. Schatz MC, Delcher AL, Salzberg SL: **Assembly of large genomes using second-generation sequencing**. *Genome Res* 2010, **20**(9):1165-1173.

11. Levy S, Sutton G, Ng PC, Feuk L, Halpern AL, Walenz BP, Axelrod N, Huang J, Kirkness EF, Denisov G *et al*: **The diploid genome sequence of an individual human**. *PLoS Biol* 2007, **5**(10):e254.

12. Mollick E: **Establishing Moore's law**. *Annals of the History of Computing, IEEE* 2006, **28**(3):62-75.

13. **Illumina Announces HiSeq(TM) 2000 Sequencing System (http://investor.illumina.com/phoenix.zhtml?c=121127&p=irol-newsArticle&ID=1374339)**. Accessed January 7, 2012.

14. **Illumina Presents Development Roadmap for Scaling its Genome Analyzer (http://investor.illumina.com/phoenix.zhtml?c=121127&p=irol-newsArticle&ID=1252407)**. Accessed January 7, 2012.

15. Stein LD: **The case for cloud computing in genome informatics**. *Genome Biol* 2010, **11**(5):207.

16. **Genomics: High-throughput "Next-Generation" Sequencing Facilities Statistics (http://pathogenomics.bham.ac.uk/hts/stats)**. Accessed January 8, 2012.

17. Ferragina P, Manzini G: **Opportunistic data structures with applications**. In*: 2000*: IEEE; 2000: 390-398.

18. Langmead B, Trapnell C, Pop M, Salzberg SL: **Ultrafast and memory-efficient alignment of short DNA sequences to the human genome**. *Genome Biol* 2009, **10**(3):R25.

19. Langmead B: **Aligning short sequencing reads with Bowtie**. *Curr Protoc Bioinformatics* 2010, **Chapter 11**:Unit 11 17.

20. Li H, Homer N: **A survey of sequence alignment algorithms for next-generation sequencing**. *Brief Bioinform* 2010, **11**(5):473-483.

21. Darling A, Carey L, Feng W: **The design, implementation, and evaluation of mpiBLAST**. *Proceedings of ClusterWorld* 2003, **2003**.

22. Jackson BG, Schnable PS, Aluru S: **Parallel short sequence assembly of transcriptomes**. *BMC Bioinformatics* 2009, **10 Suppl 1**:S14.

23. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I: **ABySS: a parallel assembler for short read sequence data**. *Genome Res* 2009, **19**(6):1117-1123.

24. White T: **Hadoop: The definitive guide**: Yahoo Press; 2010.

25. **Amazon Web Services (http://aws.amazon.com)**. Accessed January 8, 2012.

26. Schatz MC: **CloudBurst: highly sensitive read mapping with MapReduce**. *Bioinformatics* 2009, **25**(11):1363-1369.

27. Dean J, Ghemawat S: **MapReduce: Simplified data processing on large clusters**. *Communications of the ACM* 2008, **51**(1):107-113.

28. Baker M: **Next-generation sequencing: adjusting to data overload**. *Nature Methods* 2010, **7**(7):495-499.

29. Schatz MC, Langmead B, Salzberg SL: **Cloud computing and the DNA data race**. *Nat Biotechnol* 2010, **28**(7):691-693.

30. Gusfield D: **Algorithms on strings, trees, and sequences: computer science and computational biology**: Cambridge Univ Pr; 1997.

31. Smith TF, Waterman MS: **Identification of common molecular subsequences**. *J Mol Biol* 1981, **147**(1):195-197.

32. Gotoh O: **An improved algorithm for matching biological sequences**. *J Mol Biol* 1982, **162**(3):705-708.

33. Li H, Ruan J, Durbin R: **Mapping short DNA sequencing reads and calling variants using mapping quality scores**. *Genome research* 2008, **18**(11):1851-1858.

34. Li H, Durbin R: **Fast and accurate short read alignment with Burrows-Wheeler transform**. *Bioinformatics* 2009, **25**(14):1754-1760.

35. Li H, Durbin R: **Fast and accurate long-read alignment with Burrows-Wheeler transform**. *Bioinformatics* 2010, **26**(5):589-595.

36. Li R, Li Y, Kristiansen K, Wang J: **SOAP: short oligonucleotide alignment program**. *Bioinformatics* 2008, **24**(5):713-714.

37. Li R, Yu C, Li Y, Lam TW, Yiu SM, Kristiansen K, Wang J: **SOAP2: an improved ultrafast tool for short read alignment**. *Bioinformatics* 2009, **25**(15):1966-1967.

38. Lin H, Zhang Z, Zhang MQ, Ma B, Li M: **ZOOM! Zillions of oligos mapped**. *Bioinformatics* 2008, **24**(21):2431-2437.

39. Burkhardt S, K√§rkk√§inen J: **Better Filtering with Gapped-Grams**. *Fundamenta informaticae* 2003, **23**:1001-1018.

40. Rumble SM, Lacroute P, Dalca AV, Fiume M, Sidow A, Brudno M: **SHRiMP: accurate mapping of short color-space reads**. *PLoS Comput Biol* 2009, **5**(5):e1000386.

41. Weese D, Emde AK, Rausch T, Doring A, Reinert K: **RazerS--fast read mapping with sensitivity control**. *Genome Res* 2009, **19**(9):1646-1654.

42. Delcher AL, Kasif S, Fleischmann RD, Peterson J, White O, Salzberg SL: **Alignment of whole genomes**. *Nucleic Acids Res* 1999, **27**(11):2369-2376.

43. Kurtz S, Phillippy A, Delcher AL, Smoot M, Shumway M, Antonescu C, Salzberg SL: **Versatile and open software for comparing large genomes**. *Genome Biol* 2004, **5**(2):R12.

44. Ghodsi M, Pop M: **Inexact Local Alignment Search over Suffix Arrays**. In*: 2009*: IEEE; 2009: 83-87.

45. Khan Z, Bloom JS, Kruglyak L, Singh M: **A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays**. *Bioinformatics* 2009, **25**(13):1609-1616.

46. Kurtz S: **Reducing the space requirement of su x trees**. *Software| Practice and Experience* 1999, **29**(13):1149-1171.

47. Burrows M, Wheeler DJ: **A block-sorting lossless data compression algorithm**. 1994.

48. Healy J, Thomas EE, Schwartz JT, Wigler M: **Annotating large genomes with exact word matches**. *Genome Res* 2003, **13**(10):2306-2315.

49. Lippert RA: **Space-efficient whole genome comparisons with Burrows-Wheeler transforms**. *J Comput Biol* 2005, **12**(4):407-415.

50. Graf S, Nielsen FG, Kurtz S, Huynen MA, Birney E, Stunnenberg H, Flicek P: **Optimized design and assessment of whole genome tiling arrays**. *Bioinformatics* 2007, **23**(13):i195-204.

51. Lam TW, Sung WK, Tam SL, Wong CK, Yiu SM: **Compressed indexing and local alignment of DNA**. *Bioinformatics* 2008, **24**(6):791-797.

52. Ferragina P, Manzini G: **An experimental study of an opportunistic index**. In*: 2001*: Society for Industrial and Applied Mathematics; 2001: 269-278.

53. Ewing B, Hillier LD, Wendl MC, Green P: **Base-calling of automated sequencer traces usingPhred. I. Accuracy assessment**. *Genome research* 1998, **8**(3):175-185.

54. Herold J, Kurtz S, Giegerich R: **Efficient computation of absent words in genomic sequences**. *BMC Bioinformatics* 2008, **9**:167.

55. Lam T, Li R, Tam A, Wong S, Wu E, Yiu S: **High throughput short read alignment via bi-directional bwt**. In*: 2009*: IEEE; 2009: 31-36.

56. Manber U, Myers G: **Suffix arrays: a new method for on-line string searches**. In*: 1990*: Society for Industrial and Applied Mathematics; 1990: 319-327.

57. Larsson NJ, Sadakane K: **Faster suffix sorting**: Citeseer; 1999.

58. Karkkainen J: **Fast BWT in small space by blockwise suffix sorting**. *Theoretical Computer Science* 2007, **387**(3):249-257.

59. Burkhardt S, K√§rkk√§inen J: **Fast lightweight suffix array construction and checking**. In*: 2003*: Springer; 2003: 55-69.

60. **A map of human genome variation from population-scale sequencing**. *Nature* 2010, **467**(7319):1061-1073.

61. Leinonen R, Sugawara H, Shumway M: **The sequence read archive**. *Nucleic Acids Res* 2011, **39**(Database issue):D19-21.

62. **Bowtie (http://bowtie-bio.sourceforge.net/index.shtml)**. Accessed January 8, 1012.

63. Bentley DR, Balasubramanian S, Swerdlow HP, Smith GP, Milton J, Brown CG, Hall KP, Evers DJ, Barnes CL, Bignell HR *et al*: **Accurate whole human genome sequencing using reversible terminator chemistry**. *Nature* 2008, **456**(7218):53-59.

64. Ley TJ, Mardis ER, Ding L, Fulton B, McLellan MD, Chen K, Dooling D, Dunford-Shore BH, McGrath S, Hickenbotham M *et al*: **DNA sequencing of**

a cytogenetically normal acute myeloid leukaemia genome. *Nature* 2008, **456**(7218):66-72.

65. Wang J, Wang W, Li R, Li Y, Tian G, Goodman L, Fan W, Zhang J, Li J, Guo Y *et al*: **The diploid genome sequence of an Asian individual**. *Nature* 2008, **456**(7218):60-65.

66. Li H, Handsaker B, Wysoker A, Fennell T, Ruan J, Homer N, Marth G, Abecasis G, Durbin R: **The Sequence Alignment/Map format and SAMtools**. *Bioinformatics* 2009, **25**(16):2078-2079.

67. Trapnell C, Pachter L, Salzberg SL: **TopHat: discovering splice junctions with RNA-Seq**. *Bioinformatics* 2009, **25**(9):1105-1111.

68. Trapnell C, Williams BA, Pertea G, Mortazavi A, Kwan G, van Baren MJ, Salzberg SL, Wold BJ, Pachter L: **Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation**. *Nat Biotechnol* 2010, **28**(5):511-515.

69. Langmead B, Hansen KD, Leek JT: **Cloud-scale RNA-sequencing differential expression analysis with Myrna**. *Genome Biol* 2010, **11**(8):R83.

70. Chen LY, Wei KC, Huang AC, Wang K, Huang CY, Yi D, Tang CY, Galas DJ, Hood LE: **RNASEQR--a streamlined and accurate RNA-seq sequence analysis program**. *Nucleic Acids Res* 2011.

71. Cumbie JS, Kimbrel JA, Di Y, Schafer DW, Wilhelm LJ, Fox SE, Sullivan CM, Curzon AD, Carrington JC, Mockler TC *et al*: **GENE-counter: a computational pipeline for the analysis of RNA-Seq data for gene expression differences**. *PLoS One* 2011, **6**(10):e25279.

72. Hansen K, Langmead, B, Irizarry, RA: **From whole genome bisulfite sequencing reads to differentially methylated regions**. *In preparation*.

73. Krueger F, Andrews SR: **Bismark: a flexible aligner and methylation caller for Bisulfite-Seq applications**. *Bioinformatics* 2011, **27**(11):1571-1572.

74. Pedersen B, Hsieh TF, Ibarra C, Fischer RL: **MethylCoder: software pipeline for bisulfite-treated sequences**. *Bioinformatics* 2011, **27**(17):2435-2436.

75. Chen PY, Cokus SJ, Pellegrini M: **BS Seeker: precise mapping for bisulfite sequencing**. *BMC Bioinformatics* 2010, **11**:203.

76. **Zorro (http://lge.ibi.unicamp.br/zorro/)**. Accessed January 8, 2012.

77. Boetzer M, Henkel CV, Jansen HJ, Butler D, Pirovano W: **Scaffolding pre-assembled contigs using SSPACE**. *Bioinformatics* 2011, **27**(4):578-579.

78. Goecks J, Nekrutenko A, Taylor J: **Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences**. *Genome Biol* 2010, **11**(8):R86.

79. Kallio MA, Tuimala JT, Hupponen T, Klemela P, Gentile M, Scheinin I, Koski M, Kaki J, Korpelainen EI: **Chipster: user-friendly analysis software for microarray and other high-throughput data**. *BMC Genomics* 2011, **12**:507.

80. Needleman SB, Wunsch CD: **A general method applicable to the search for similarities in the amino acid sequence of two proteins**. *J Mol Biol* 1970, **48**(3):443-453.

81. Needleman SB, Wunsch CD: **A general method applicable to the search for similarities in the amino acid sequence of two proteins**. *Journal of molecular biology* 1970, **48**(3):443-453.

82. Wozniak A: **Using video-oriented instructions to speed up sequence comparison**. *Computer applications in the biosciences: CABIOS* 1997, **13**(2):145-150.

83. Rognes T, Seeberg E: **Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors**. *Bioinformatics* 2000, **16**(8):699.

84. Farrar M: **Striped Smith-Waterman speeds database searches six times over other SIMD implementations**. *Bioinformatics* 2007, **23**(2):156-161.

85. Rognes T: **Faster Smith-Waterman database searches with inter-sequence SIMD parallelisation**. *BMC Bioinformatics* 2011, **12**:221.

86. **Genome Reference Consortium: human (http://www.ncbi.nlm.nih.gov/projects/genome/assembly/grc/human/index.shtml)**. Accessed January 11, 2012.

87. Ajay SS, Parker SC, Ozel Abaan H, Fuentes Fajardo KV, Margulies EH: **Accurate and comprehensive sequencing of personal genomes**. *Genome Res* 2011.

88. Rothberg JM, Hinz W, Rearick TM, Schultz J, Mileski W, Davey M, Leamon JH, Johnson K, Milgrew MJ, Edwards M *et al*: **An integrated semiconductor device enabling non-optical genome sequencing**. *Nature* 2011, **475**(7356):348-352.

89. Holtgrewe M: **Mason,Äìa read simulator for second generation sequencing data**. *Technical Report FU Berlin* 2010.

90. Wu TD, Nacu S: **Fast and SNP-tolerant detection of complex variants and splicing in short reads**. *Bioinformatics* 2010, **26**(7):873-881.

91. **Mosaik (http://bioinformatics.bc.edu/marthlab/Mosaik)**. Accessed January 12, 2012.

92. David M, Dzamba M, Lister D, Ilie L, Brudno M: **SHRiMP2: sensitive yet practical SHort Read Mapping**. *Bioinformatics* 2011, **27**(7):1011-1012.

93. **Amazon Elastic Compute Cloud (http://aws.amazon.com/ec2/)**. Accessed January 9, 2012.

94. Langmead B, Schatz MC, Lin J, Pop M, Salzberg SL: **Searching for SNPs with cloud computing**. *Genome Biol* 2009, **10**(11):R134.

95. **Welcome to Apache™ Hadoop™ (http://hadoop.apache.org/)**. Accessed January 10, 2012.

96. Li R, Li Y, Fang X, Yang H, Wang J, Kristiansen K: **SNP detection for massively parallel whole-genome resequencing**. *Genome Res* 2009, **19**(6):1124-1132.

97. **Amazon Elastic MapReduce (http://aws.amazon.com/elasticmapreduce/)**. Accessed January 9, 2012.

98. **Sort Benchmark Home Page (http://sortbenchmark.org/)**. Accessed January 9, 2012.

99. Frazer KA, Ballinger DG, Cox DR, Hinds DA, Stuve LL, Gibbs RA, Belmont JW, Boudreau A, Hardenbol P, Leal SM *et al*: **A second generation human haplotype map of over 3.1 million SNPs**. *Nature* 2007, **449**(7164):851-861.

100. **YanHuang Project :: YH1 Genome Database (http://yh.genomics.org.cn/)**. Accessed January 9, 2012.
101. **Amazon Simple Storage Service (http://aws.amazon.com/s3/)**. Accessed January 10, 2012.
102. Hansen KD, Wu Z, Irizarry RA, Leek JT: **Sequencing technology does not eliminate biological variability**. *Nature Biotechnology* 2011, **29**(7):572-573.
103. Bullard JH, Purdom E, Hansen KD, Dudoit S: **Evaluation of statistical methods for normalization and differential expression in mRNA-Seq experiments**. *BMC Bioinformatics* 2010, **11**:94.
104. Anders S, Huber W: **Differential expression analysis for sequence count data**. *Genome Biol* 2010, **11**(10):R106.
105. Birol I, Jackman SD, Nielsen CB, Qian JQ, Varhol R, Stazyk G, Morin RD, Zhao Y, Hirst M, Schein JE *et al*: **De novo transcriptome assembly with ABySS**. *Bioinformatics* 2009, **25**(21):2872-2877.
106. Grabherr MG, Haas BJ, Yassour M, Levin JZ, Thompson DA, Amit I, Adiconis X, Fan L, Raychowdhury R, Zeng Q: **Full-length transcriptome assembly from RNA-Seq data without a reference genome**. *Nature Biotechnology* 2011, **29**(7):644-652.
107. Guttman M, Garber M, Levin JZ, Donaghey J, Robinson J, Adiconis X, Fan L, Koziol MJ, Gnirke A, Nusbaum C *et al*: **Ab initio reconstruction of cell type-specific transcriptomes in mouse reveals the conserved multi-exonic structure of lincRNAs**. *Nat Biotechnol* 2010, **28**(5):503-510.
108. Pickrell JK, Marioni JC, Pai AA, Degner JF, Engelhardt BE, Nkadori E, Veyrieras JB, Stephens M, Gilad Y, Pritchard JK: **Understanding mechanisms underlying human gene expression variation with RNA sequencing**. *Nature* 2010, **464**(7289):768-772.