ABSTRACT

Title of Document: Highly Scalable Short Read Alignment with the Burrows-Wheeler Transform and Cloud Computing

Benjamin Langmead, Master of Science, 2009

Directed By: Professor Steven L. Salzberg
Department of Computer Science

Improvements in DNA sequencing have both broadened its utility and dramatically increased the size of sequencing datasets. Sequencing instruments are now used regularly as sources of high-resolution evidence for genotyping, methylation profiling, DNA-protein interaction mapping, and characterizing gene expression in the human genome and in other species. With existing methods, the computational cost of aligning short reads from the Illumina instrument to a mammalian genome can be very large: on the order of many CPU months for one human genotyping project. This thesis presents a novel application of the Burrows-Wheeler Transform that enables the alignment of short DNA sequences to mammalian genomes at a rate much faster than existing hashtable-based methods. The thesis also presents an extension of the technique that exploits the scalability of Cloud Computing to perform the equivalent of one human genotyping project in hours.

HIGHLY SCALABLE SHORT READ ALIGNMENT WITH THE BURROWS-WHEELER TRANSFORM AND CLOUD COMPUTING.

By

Benjamin Thomas Langmead.

Thesis 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

Master of Science
2009

Advisory Committee:

Professor Steven L. Salzberg, Chair
Professor Mihai Pop
Professor Carl Kingsford

Dedication

To Sara.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Bowtie

Improvements in DNA sequencing have both broadened its applications and dramatically increased the size of sequencing datasets. Technologies from Illumina and Applied Biosystems have been used to profile methylation patterns (Methyl-Seq) [1], to map DNA-protein interactions (ChIP-Seq) [2], and to identify differentially expressed genes and novel splice junctions (RNA-Seq) [3] in the human genome and other species. The Illumina instrument was recently used to re-sequence three human genomes, one from a cancer patient and two from previously unsequenced ethnic groups [4-6]. Each of these studies required the alignment of large numbers of short DNA sequences, "short reads," onto the human genome. For example, two of the studies [4, 5] used the short read alignment tool Maq [7] to align more than 130 billion bases (about 45× coverage) of short Illumina reads to a human reference genome in order to detect genetic variations. The third human re-sequencing study [6] used the SOAP program [8] to align more than 100 billion bases to the reference genome. In addition to these projects, the 1,000 Genomes project is in the process of using high-throughput sequencing instruments to sequence a total of about six trillion base pairs of human DNA [9].

The computational cost of aligning many short reads to a mammalian genome using the computational methods employed by Maq and SOAP is very large. For example, extrapolating from the results presented here in Tables 1 and

2, Maq would require more than 5 CPU-months and SOAP more than 3 CPU-years to align the 140 billion bases from the study by Ley and coworkers [5]. Although using Maq or SOAP for this purpose is feasible by parallelizing the work across many CPUs, there is a clear need for new methods that can keep pace with the growing throughput of these instruments while making more economical use of computational resources.

Maq and SOAP take a similar basic algorithmic approach to many other recent read mapping tools such as RMAP [10], ZOOM [11], and SHRiMP [12]. These tools build a hash table of short oligomer subsequences present in either the reads (SHRiMP, Maq, RMAP, and ZOOM) or the reference (SOAP). Some employ recent theoretical advances to align reads quickly without sacrificing sensitivity. For example, ZOOM uses "spaced seeds" to significantly outperform RMAP, which is based on a simpler algorithm due to Baeza-Yaetes and Perleberg [13]. Spaced seeds have been shown to yield higher sensitivity than contiguous seeds of the same length [14, 15]. SHRiMP employs a combination of spaced seeds and the Smith-Waterman [16] algorithm to align reads with high sensitivity at the expense of speed. Eland is a commercial alignment program available from Illumina that also uses a hash-based spaced-seed algorithm to align reads.

Chapters 2-4 of this thesis present Bowtie, a short read aligner that exploits an indexing strategy similar to the Full-text Minute-space (FM) Index [17, 18] to achieve ultrafast and memory-efficient alignment of short reads to mammalian genomes. In experiments using reads from the 1,000 Genomes project, Bowtie aligns 35-base reads to the human genome at a rate of more than

25 million reads per CPU-hour, which is more than 35 times faster than Maq and 300 times faster than SOAP under the same conditions (see Tables 1 and 2). A typical memory footprint for the Bowtie aligner is 2.2 gigabytes for single-end alignment and 2.9 gigabytes for paired end, small enough to run on a workstation with 4 gigabytes of physical memory. Bowtie also supports a mode where the memory footprint is limited to about 1.1 gigabytes, at the expense of some efficiency and features. The space-efficiency of the Bowtie index is such that indexes can be distributed over the Internet and stored on disk for re-use.

Bowtie was conceived and implemented primarily by the author, with help from Cole Trapnell, Mihai Pop and Steven L. Salzberg. A paper describing the Bowtie tool appeared in the journal Genome Biology [19].

Crossbow

With the advent of robust implementations of cloud computing software and services such as Hadoop [20, 21] and Amazon Web Services [21], it is increasingly possible to solve data-intensive problems efficiently without having to own and maintain a large pool of computer equipment. At the same time, DNA sequencing instruments are now producing enough high-resolution data to regularly resequence entire human genomes. Cases in point are three papers that appeared in the Nov. 6, 2008 issue of Nature [4-6], and the 1000 Genomes Project, to which aims to sequence a total of about six trillion base pairs of human DNA [9].

Crossbow is a software tool that combines the Bowtie aligner with a simple variant caller, both customized to work together in a MapReduce [22]

context. Crossbow runs in any environment that supports the Hadoop MapReduce implementation, including Hadoop-enabled virtual machines provided by Amazon for use within the Elastic Compute Cloud (EC2) web service. Chapter 5 of this thesis presents the design of Crossbow along with results demonstrating that it is capable of aligning about 14.3×-coverage worth of human Illumina reads in 1 hour and 11 minutes using an EC2 cluster of 20 Extra-Large High-CPU nodes, incurring a total of about $32 in cluster rental fees. Since Amazon EC2 is available to anyone with an Amazon AWS account, the technique and the results achieved are easily reproducible by others.

The author performed the work described in Chapter 5. Ongoing work on Crossbow is a collaboration between the author and Michael Schatz.

Chapter 2: The Burrows-Wheeler Transform and FM Index

This chapter establishes the theoretical underpinnings of the Burrows-Wheeler Transform and the FM Index and describes how the FM Index potentially represents an improvement over indexing techniques used previously for short read alignment.

Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) of a text is a reversible permutation of its characters. Originally developed within the context of data compression, BWT-based indexing allows large texts to be searched efficiently in a small memory footprint. It has been applied previously to bioinformatics applications including oligomer counting [23], whole-genome alignment [24], tiling microarray probe design [25], and Smith-Waterman alignment to a large reference [26].

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) (right) from the input text T (left) via the Burrows-Wheeler Matrix of T (center).

Burrows-Wheeler Matrices have a property called the *Last First* (*LF*) *Mapping*. The property is: the i^{th} occurrence of character *c* in the last column corresponds to the same text character as the i^{th} occurrence of *c* in the first column. Burrows and Wheeler [27] 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 by their second character. Consider just the rows of M' beginning with character *c*. These rows must appear in lexicographical order with respect to each other, since 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 whose first column contains 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 precedes that character in the text (cyclically). 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 $, which is always the first row), we 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 Algorithm 1 below. 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 Algorithm 2 below. Figure 2 illustrates how the UNPERMUTE algorithm reconstructs the original string "acaacg$" from the permuted string "gc$aaac".

Algorithm 1 LF(r)

$c \Leftarrow BWT[r]$
return $C[c] + \text{Occ}(c, r) + 1$

Algorithm 2 UNPERMUTE

```
r ⇐ 1
T ⇐ " "
while BWT[r] ≠ $ do
    T ⇐ prepend BWT[r] to T
    r ⇐ LF(r)
end while
return T
```

Figure 2 Steps taken by the UNPERMUTE algorithm to recreate the original text T from the Burrows-Wheeler Transformed text BWT(T).

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 appear consecutively, i.e., within a contiguous range of rows. The EXACTMATCH algorithm (Algorithm 3 below) 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 Algorithm 4 below. Figure 3 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 the paper by Ferragina and Manzini [17].

Algorithm 3 LFC(r, c)

1: **return** $C[c] + \text{Occ}(c, r) + 1$

Algorithm 4 EXACTMATCH$(P[1, p])$

1: $c \Leftarrow P[p]$
2: $sp \Leftarrow C[c] + 1$
3: $ep \Leftarrow C[c + 1] + 1$
4: $i \Leftarrow p - 1$
5: **while** $sp < ep$ and $i \geq 1$ **do**
6: $\quad c \Leftarrow P[i]$
7: $\quad sp \Leftarrow LFC(c, sp)$
8: $\quad ep \Leftarrow LFC(c, ep)$
9: $\quad i \Leftarrow i - 1$
10: **end while**
11: **return** sp, ep

Figure 3 Steps taken by the EXACTMATCH algorithm to match the pattern "aac" in the text "acaacg". Successive pairs of red arrows delimit ranges of Burrows-Wheeler rows prefixed by increasingly longer suffixes (also red) of the query string.

Making EXACTMATCH efficient with checkpointing

It remains unclear whether UNPERMUTE and EXACTMATCH scale well to large texts. The 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.

Mapping rows to reference offsets

The final output of the EXACTMATCH algorithm is a range of matrix rows beginning with a given query string P (delimited by the *sp* and *ep* variables from Algorithm 4). 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 naïve 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 (0-based) 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 4. 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.

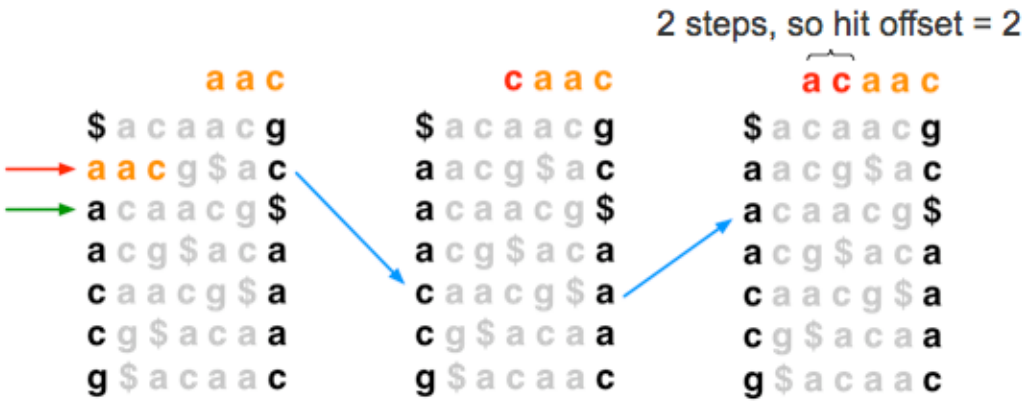

Figure 4 One naïve solution for resolving a row's reference offset: repeatedly apply rule r = LF(r) until the row with $ in the last column is reached. The number of calls made to LF(r) is the row's 0-based reference offset.

Another naïve solution is to, at index building time, simply pre-compute and store an array parallel to BWT(T) containing the reference offsets of each row. This array is equivalent to the suffix array of T. To resolve the reference offset of row r, we simply look up element r in the pre-calculated array (see Figure 5). 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 $n \log_2(n)$, which, for the approximately 3-gigabase human genome, necessitates about 12 gigabytes of storage.

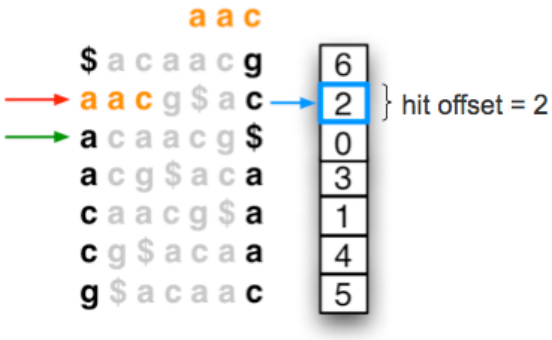

Figure 5 Another naïve solution for resolving the reference offset for a given row: pre-calculate the offset for every row and store the pre-calculated offsets in an array parallel to BWT(T). This requires as much memory as the suffix array of T.

Ferragina and Manzini [17] propose a hybrid scheme whereby a subset of the rows of the matrix are "marked" with pre-calculated text offsets. To retrieve a row *r*'s text offset, we first check if *r* is marked. If so, we report *r*'s pre-calculated text offset. If not, we calculate *r* = LF(*r*) repeatedly until *r* becomes a marked row, at which point we report the pre-calculated offset for r plus the number of times LF(*r*) was called before reaching *r*. Marking a larger fraction of the rows allows text offsets to be calculated faster on average, but marking a smaller fraction reduces the overall size of the index. Bowtie adopts this scheme with a default (but configurable) policy of marking every 32nd row. Figure 6 illustrates an example where LF(*r*) is called exactly once, causing the walk to enter a row marked with offset 1.

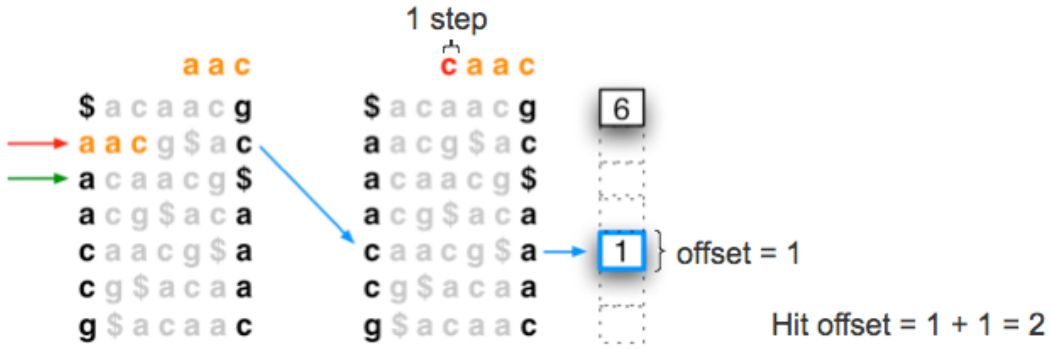

Figure 6 An example of Ferragina and Manzini's hybrid scheme where the offset is calculated by calling *r* = LF(*r*) zero or more times (once in this case) to reach a marked row. The final 0-based result is the sum of the number of calls to LF(*r*) required to reach the marked row plus the marked row's pre-calculated offset.

Bowtie's scheme of marking every 32nd row does not provide a sublinear worst-case guarantee 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 rows to mark 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 because Ferragina and Manzini's is more complex to implement and because the expected number of calls to LF (as opposed to the worst-case number) is not very different between the two.

The FM Index is small

The major components of an FM Index include the string BWT(T), the checkpoint data, and the data structure encoding the mapping between marked rows and their corresponding reference offsets. Bowtie's version of the FM Index stores a few additional structures, including, for example, a lookup table used to quickly calculate the matrix range corresponding to a given 10-mer.

Under Bowtie's default settings, the largest single component of the FM Index is the BWT(T) sequence. Bowtie stores BWT(T) in a 2-bit-per-base format, causing it to occupy about 680 megabytes in the case of the assembled human genome. The checkpoints occupy another approximately 14% the space of the packed BWT(T) string, assuming Bowtie's default policy of storing a checkpoint every 448 BWT characters. The text offsets for marked rows occupy about 50% the space of the packed BWT(T) string, assuming Bowtie's default policy of marking every 32nd row. With some other small structures, the overall FM Index for a given genome is about 65-70% larger than the packed BWT(T) string alone. A Bowtie FM Index for the assembled human genome sequence occupies about 1.1 gigabytes. In practice, Bowtie often keeps two FM Indexes resident in memory at a time, for reasons discussed later. Thus, the total memory

footprint is about 2.2 gigabytes in practice. When the reference string is also resident in memory (i.e. for paired-end alignment), the total memory footprint is about 2.9 gigabytes.

The FM for the human genome is significantly smaller than what can be achieved with other indexing techniques such as suffix trees, suffix arrays, and k-mer hashtables. All of these alternatives require at least 4 bytes per character for the human genome. Some, like the suffix tree, require much more (a compact suffix-tree representation described by Kurtz in 1999 [28] reports a ratio of 20 bytes per base).

A compact indexing scheme like the FM Index removes the need for some of the painful compromises made by tools with larger indexes. Maq, for instance, uses a spaced-seed k-mer hashtable, but, perhaps because such a hashtable calculated over the reference genome would occupy roughly 12 gigabytes of RAM, it chooses to index the reads rather than the genome. This is a significant compromise since it requires the user to provide reads to Maq in batches and Maq must then scan the reference three times per batch to perform seed-and-extend. Also, since an index calculated over reads is relevant only with respect to that batch of reads, it is not as re-usable as a genome index. SOAP also uses a spaced-seed k-mer hashtable, but it indexes the genome rather than the reads. However, the size of the hashtable is such that SOAP requires a computer with about 16 gigabytes of RAM to align reads to the human genome. The larger the index, the more significant the compromises made. In contrast, Bowtie runs effectively on a computer with 4 gigabytes of RAM. A Bowtie index can be re-used across any

number of batches of reads, and each read need only be aligned against a single index.

Chapter 3: Short read alignment atop the FM Index

This chapter describes how the Bowtie aligner uses search to build the FM Index's exact-matching facility into an inexact-matching facility. The overall search strategy and the associated pruning strategies are described in detail, along with some examples of particular strategies used to enforce alignment policies. Other significant features of the Bowtie aligner are also described. Finally, empirical performance results using human read data from the Short Read Archive are presented.

Adding inexactness to EXACTMATCH

EXACTMATCH is not sufficient for aligning genomic short reads because the best local alignment of a read may contain differences. Such differences may be due to sequencing errors, genuine 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, extensive pruning is possible in practice. Pruning the space to include only those paths with at most 2 mismatches, for example, yields a space that is

quadratic in the length of the read[1]. We find that the method described is generally tractable for up to (at least) three mismatches in practice. Though the technique can be made to deal with gaps as well as mismatches, the Bowtie implementation currently deals only with mismatches. Thus, only mismatches will be considered in the subsequent discussion.

The search makes use of numeric quality values on the Phred scale [29], where, if the sequencing instrument predicts that the probability of a base having been miscalled is p, the Phred quality value of the base is reported as $-10 \log_{10} p$. 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 most likely candidates first.

Inexact search proceeds similarly to EXACTMATCH, calculating Burrows-Wheeler ranges for successively longer query suffixes. The difference is that when the search arrives at an empty Burrows-Wheeler range (indicating that the corresponding suffix does not occur in the text), the algorithm may select a previously examined query position and substitute a different base there, introducing a hypothetical mismatch into the alignment at that position. This is called a "backtrack." After executing a backtrack, the EXACTMATCH search

---

[1] The number of unpruned edges in a search space for all alignments of a read of length n with at most 2 mismatches assuming no pruning besides policy pruning is $\frac{1}{2}(9n^2 - 21n + 14)$. The same for alignments with at most 1 mismatch is $\frac{1}{2}(3n^2 - n)$.

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 is futile, since the emptiness of the range implies that no alignments of the hypothesized type are possible in the reference. This is called "empty-range pruning." Figure 7 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 practice, maximizing the amount of pruning possible is critical to minimizing the running time of the search.

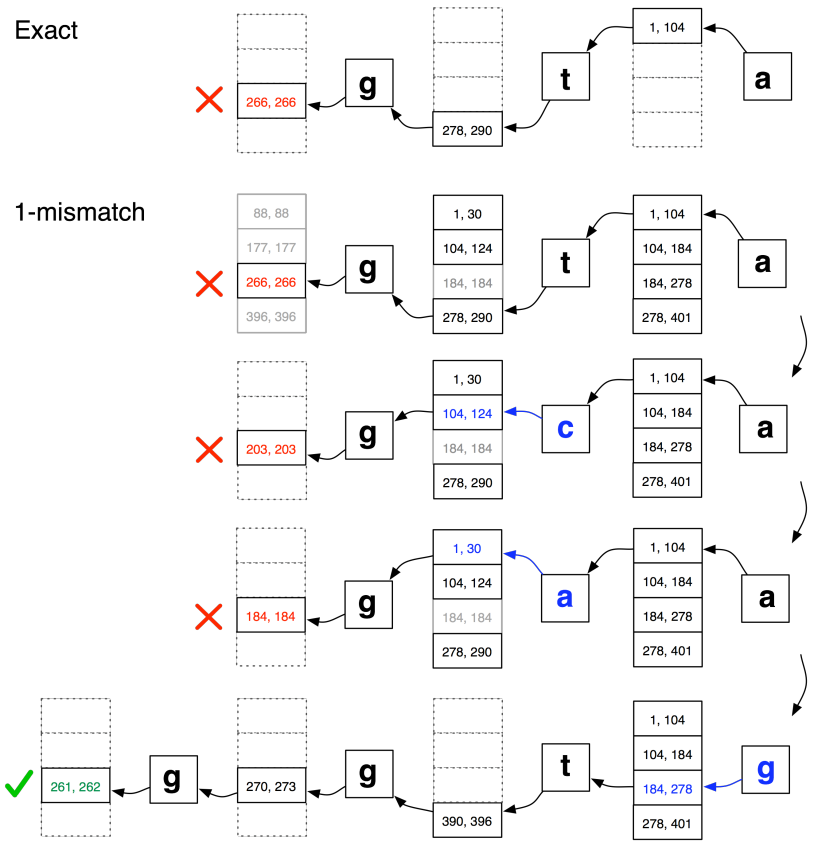

Figure 7 Example of how exact and 1-mismatch algorithms might proceed. In this case, the query "ggta" does not have an exact match in the text, but does have a 1-mismatch alignment where a "g" in the reference is substituted for "a" in the read. Pairs of numbers represent the *sp*, *ep* pairs calculated in an iteration of EXACTMATCH (Algorithm 4). 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.

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 so many backtracks that performance is adversely affected. Since relatively 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 coincidence, excessive backtracking is

particularly prevalent in first several levels 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 [30]). If no such alignment exists, the search is forced to explore all combinations of 2 backtracks within that first ten levels of the search space. A lower bound on the total number of backtracks performed in this case is 435[2].

Bowtie mitigates excessive backtracking using "double indexing." With double indexing, two Burrows-Wheeler indexes of the genome are created: one indexing the normal genome sequence, called the "forward index," and a second indexing 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

---

[2] The worst-case number of backtracks in a stretch of k bases in an alignment where the alignment policy allows 2 mismatches and no empty-range pruning is possible is given by: $3k + 9 \times ½ \times (k^2 - k)$

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 8 illustrates the two cases. By forbidding backtracks to positions close to the right-hand side of the alignment, this strategy avoids a great deal of backtracking.

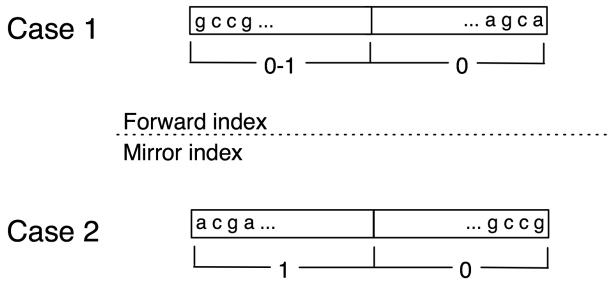

Figure 8 Two cases considered by Bowtie searching for 1-mismatch alignments. Numbers shown below the alignment segments indicate permitted numbers of substitutions in those segments.

As an implementation matter, Bowtie translates valid alignments obtained in the mirror index back into the forward coordinate system before reporting them to the user.

The technique as described so far does not specify whether cases are explored one-after-the-other (and, if so, in what order they are explored) or whether they are explored concurrently. These are implementation issues considered in later sections.

It is important to note that excessive backtracking is still not entirely eliminated with double indexing. Using the search strategy just described, 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, we find that double indexing leads to good performance in practice.

Phased 2-mismatch search

Excessive backtracking is 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 have a drastic impact on 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 9 illustrates these cases. Any given 2-mismatch alignment can be uniquely assigned to one of these three cases.

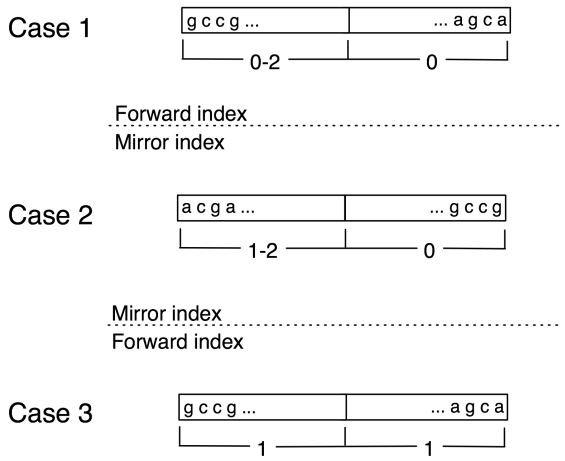

Figure 9 Three cases considered by Bowtie searching for 2-mismatch alignments. Numbers shown below the alignment segments indicate number of allowed substitutions in those segments.

The 3-case approach is relatively simple and 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 we find that while the overhead of excessive backtracking is onerous for some reads, the overall running time of the search across many reads is good in practice. The addition of more cases could improve results further; exploring more complex case decompositions is future work.

Phased 3-mismatch search

3-mismatch proceeds similarly to 2-mismatch search, but with one additional case. 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 3. 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 1-3 mismatches. Case 3 uses the forward index and constrains the right half to contain exactly 1

mismatch each and the left half to contain 1 or 2 mismatches. Case 4 uses the mirror index (and the reversed read) and constraints the right half to contain exactly 1 mismatch and the left half to contain exactly 2 mismatches. Figure 10 illustrates these cases. Any given 3-mismatch alignment can be uniquely assigned to one of the four cases

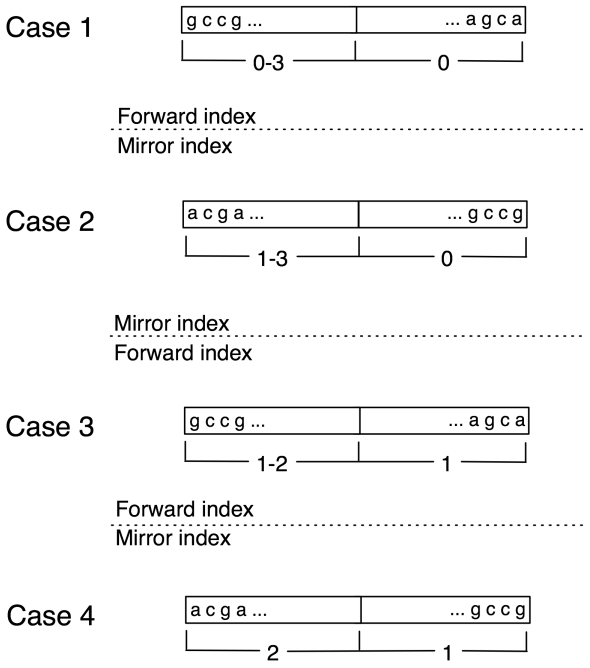

Figure 10 Four cases considered by Bowtie searching for 3-mismatch alignments. Numbers shown below the alignment segments indicate number of allowed substitutions in those segments.

Excessive backtracking is still worse a problem in the 3-mismatch case than in the 2-mismatch case for the same reasons that 2-mismatch is worse than 1-mismatch. Still, we find that overall performance of the search is still acceptable in practice.

Maq-like search

The search strategies described so far handle alignment policies where 1-3 mismatches are permitted in the entire alignment and quality values are not considered. The family of alignment policies enforced by Maq [7] is different in two ways. First, Maq enforces a ceiling on the sum of the Phred 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 bases of the read (the "seed"), not in the entire alignment. Maq permits up to two mismatches in the first 28 bases of the read by default. Any number of mismatches is permitted outside the seed, though 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.

Strategies presented in previous sections (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.

The set of cases used by Bowtie to align reads using the Maq-like alignment policy for 0-seed-mismatches, 1-seed-mismatch, 2-seed-mismatches,

and 3-seed-mismatches are shown in the following Figures 11-14. Dashed lines highlight instances where the seed portion of the alignment is re-matched in order to bridge the gap between forward and mirror indexes.

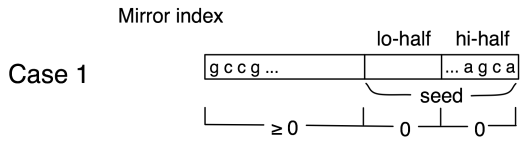

Figure 11 One case explored by Bowtie for 0-mismatch Maq-like alignment.

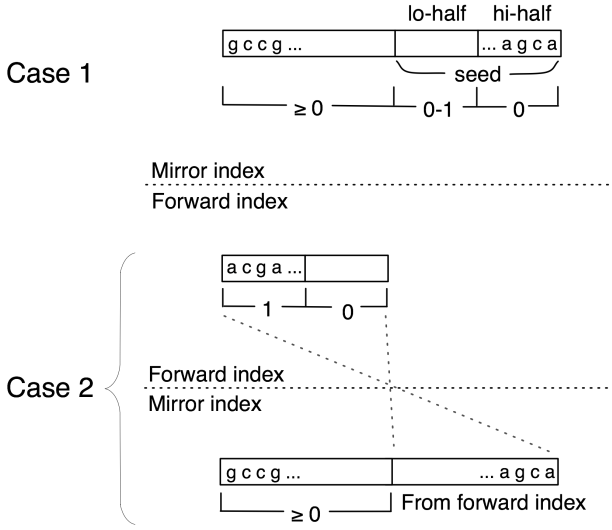

Figure 12 Two cases explored by Bowtie for 1-mismatch Maq-like alignment.

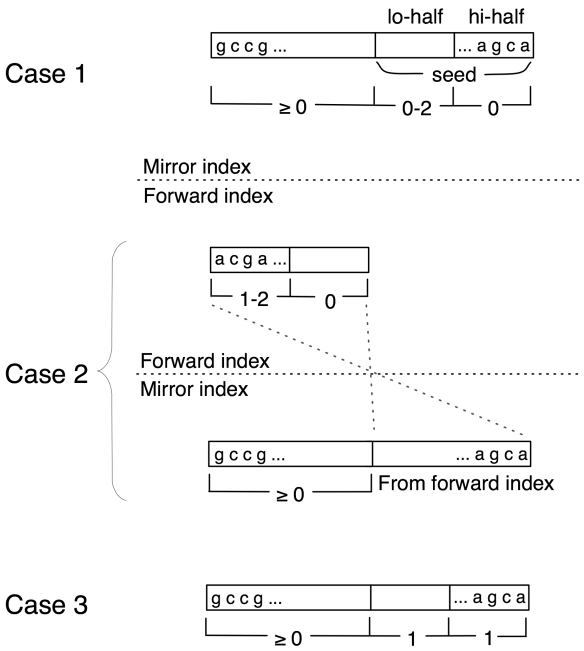

Figure 13 Three cases explored by Bowtie for 2-mismatch Maq-like alignment.

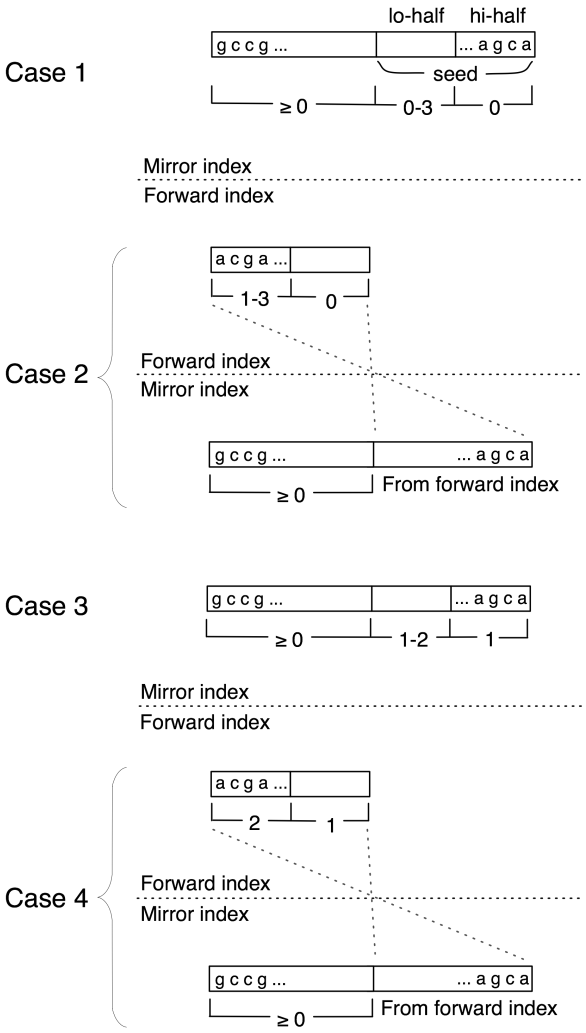

Figure 14 Four cases explored by Bowtie for 3-mismatch Maq-like alignment.

Backtracking limit

Even with the above measures, we observe that 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. We mitigate this cost 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 we expect that 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.

Phased versus interleaved search

So far, search strategies have been expressed in terms of a handful of cases where each case defines a constrained search. There is still the question of how or in what order the searches are conducted. One strategy is to explore each case's space completely, case-by-case, stopping as soon as the desired alignment(s) have been reported or all cases have been exhausted. This is a "phased" strategy. Phased search has at least two advantages: it is simple, and it potentially obviates the need to store both the forward and mirror indexes in memory at all times. Exploiting the latter can drastically reduce alignment memory footprint.

To see how memory footprint can be reduced, consider two ways of organizing the alignment computation for a given set of reads: (a) for each read, conduct a phased search for that read, or (b) for each case in the search strategy, iterate through all reads and conduct a search for that read/case combination. Option (a) is "read-outer" phased search (since the "outer loop" is a loop over reads), while option (b) is "case-outer" phased search. If only one index resides in memory at a time, read-outer phased search is very inefficient. The problem is that a given read's phased search may require one or more turnovers between the forward and mirror indexes. The 4-case 3-mismatch Maq-like search, for instance, could trigger up to four index turnovers per read. For a human index, a

turnover involves loading more than 1 gigabyte of data from the disk to memory; thus, the overhead incurred is unacceptably large. In contrast, case-outer phased search requires only a handful of index turnovers overall, irrespective of the number of reads. The 4-case 3-mismatch Maq-like search, for instance, requires only four index turnovers total. Thus, memory savings is achievable at a reasonable cost as long as the alignment computation is organized as a case-outer phased search.

Phased search also has disadvantages. If the best possible alignment has at least one mismatch, a phased search is not guaranteed to encounter that alignment before other, less optimal alignments. Consider a phased 3-case 2-mismatch Maq-like search where the best alignment has a single mismatch in the left half of the seed, but the second-best alignment has two mismatches in the right half of the seed. A phased search that follows the three cases of Figure 13 in order will encounter the second-best before the best. This is an inherent drawback of phased search combined with double indexing.

Case-outer phased search has an additional drawback. Depending on the type of alignment(s) requested by the user, the search may have to maintain a substantial amount of state across iterations of the outer case loop. For instance, if the user requests at most one alignment per read, the case-outer phased search routine must maintain sufficient state to know which reads are "done" (have already had an alignment reported) to avoid reporting extraneous alignments. For Maq-like search, where it is possible for a seed alignment to be identified in one case and then extended in the next, the search routine must persist the set of all

seed alignments for all reads from one case to the next so that they can be extended. This overhead quickly becomes unwieldy.

Early versions of Bowtie used case-outer phased search in order to exploit the memory savings, but Bowtie has since moved toward unphased or "interleaved" search, though phased search can still be enabled via the Bowtie aligner's –z/--phased option. "Interleaved" refers to any strategy where cases are not necessarily explored one-after-the-other. Rather, many cases can be explored concurrently and each individual case search can be resumed and suspended. Best-first search (described below) is interleaved in order to guarantee that it only ever extends the lowest-cost path currently active in any of the cases. The memory savings that was possible with case-outer phased search is not generally possible with interleaved search.

Depth-first search

Separate from the question of whether the search is phased or interleaved is the question of how the search space for a given case is explored. Bowtie implements two strategies: greedy, quality-guided depth-first search, and quality-guided best-first search. In the depth-first search, Bowtie only ever extends the deepest active path through the space, operating on the principle that deep paths are more likely to result in valid alignments sooner than shallow paths. When there are multiple ways to extend a path, the greedy depth-first search greedily extends in whichever way yields the lowest cumulative cost. When backtracks occur, the position at which the next substitution is hypothesized is selected according to a greedy, quality-aware heuristic. The heuristic selects from among

the positions to the left of the last hypothesized substitution (exclusive) and to the right of the position that triggered the backtrack (inclusive). Within that interval, it selects the leftmost position having the minimal quality value of the interval. Previously exhausted subtrees of the search space are marked as done so that they are not re-visited. See Figure 15 for an example of how a greedy depth-first strategy could proceed.

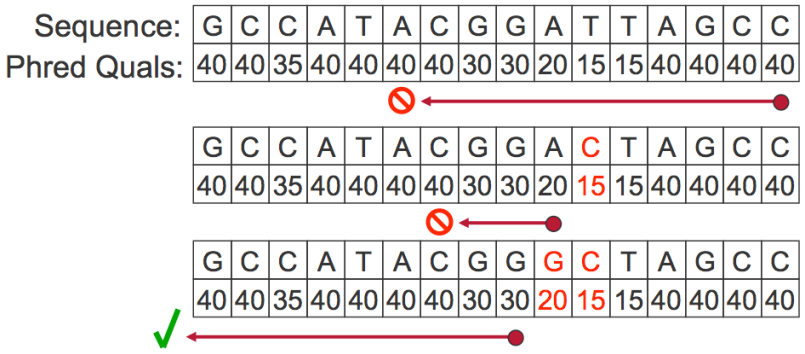

Figure 15 A greedy depth-first search for a read having a 2-mismatch alignment. Purple lines show the progress of EXACTMATCH along the length of the read, and red circles show positions where the Burrows-Wheeler range becomes empty, triggering a backtrack. Positions shown in red are where substitutions have been hypothesized. When backtracking, the search backtracks to the leftmost just-visited position with minimal quality value.

The greedy strategy has the advantage that its implementation can be simple (e.g. a recursive function), and it generally arrives at valid alignments quickly. Its usefulness, however, is limited by the fact that alignments are not necessarily encountered in best-to-worst order. If a user demands the best alignment, a depth-first search is forced to buffer potentially reportable alignments until all opportunities to find better alignments have been provably exhausted or until buffered alignments are superseded by better alignments. Another potential drawback is that the recursive-function implementation of the

greedy strategy is difficult to adapt to an interleaved context, since recursive functions cannot (straightforwardly) be suspended and resumed.

Quality-guided, greedy depth-first search is the default search mode for Bowtie, both in Bowtie's default mode and phased mode (-z/--phased option). Results show that it is generally about 1 to 2.5 times faster than best-first search using real human data, though it provides fewer guarantees regarding the quality of the alignments reported.

Best-first search and top-level best-first search

Depth-first search is efficient, but it cannot make the desirable guarantee that valid alignments are encountered in best-to-worst order. Best-first search does make this guarantee. It does so by only ever extending the path through the overall search space with the lowest cumulative cost. Path costs are calculated according to the alignment policy. If the alignment policy allows up to 3 mismatches in the entire alignment, then the cost of a path equals the number of mismatches introduced along it so far. If the alignment policy allows up to 2 mismatches in the seed portion of the alignment and enforces an overall quality ceiling of 70, then the cost of a node equals the pair (m, q), where m equals the number of mismatches introduced in the seed so far, and q equals the total of the Phred qualities of all mismatched positions (both seed and non-seed) introduced so far. Note that whether m is more or less significant than q when comparing two costs depends on the definition of "best." Bowtie assumes that m is more significant than q.

Whereas depth-first search can be implemented with a recursive function, best-first search demands a more complex implementation. Best-first requires a mechanism for suspending and resuming its progress along valid paths, since valid paths may alternate between being best and non-best throughout the search. Also, mechanisms are required to sort valid paths according to their cost (e.g. with a min heap), to remove from consideration paths that are no longer valid, and to insert new paths as they become relevant. These are all implemented in Bowtie.

Best-first search guarantees that valid alignments within a given case are encountered in best-to-worst order, but the overall search space is partitioned into many distinct per-case spaces. The best-first principle must be extended to operate among as well as within cases. Bowtie addresses this with a "top-level" best-first search that coordinates the per-case best-first searches in a way that maintains the overall guarantee.

To this end, each per-case best-first search maintains a variable "minCost" that always holds the cost of the current lowest-cost alignment that could possibly emerge from that space in the future. Note that this is not necessarily equal to the cost of the lowest-cost path active in the search. To see why, consider Case 2 for the 1-mismatch policy depicted in Figure 8. Any alignment emerging from that per-case space is bound to have at least one mismatch in the left half, even if the current cumulative low-cost path in that space does not yet include a mismatch. minCost is therefore equal to the maximum of (a) the cost of the current lowest-cost path and (b) the projected minimum-cost alignment given the constraints imposed by the case (and the read's quality values).

The top-level search instructs a per-case search to make progress by calling a per-case "advance" function. By reading each case's minCost variable, the top-level search can maintain a min heap that organizes per-case searches according to which should be advanced next. The top-level search only ever advances the per-case search that resides at the top of the heap (i.e. with the lowest minCost). When the advance function for a case is called by the top-level search, it resumes exploring the case's search space until a valid range is encountered or until the minCost variable changes, or both. If a valid range is returned by the per-case search, the top-level search simply returns it. If the minCost variable changes, the top-level search removes and then re-inserts the per-case search in the top-level min heap.

Bowtie's best-first search mode is implemented as described, and is enabled with the --best option. In practice, best-first search is usually about 1 to 2.5 times slower than greedy depth-first search, though unlike depth-first search it makes a strong guarantee about the quality of the alignments reported.

Considering the reverse-complement reference strand

The reference index is built using the forward reference strand only. Bowtie aligns the reverse-complement of the read to the forward reference strand as a proxy for aligning the original read to the reverse-complement reference strand. Handling the reverse-complement of the read requires adding a new, parallel set of reverse-complement cases to each of the strategies outlined above. Each new case corresponds to one old case, but uses the reverse-complement of the read rather than the original read. Reverse-complementing the read in the

context of a Maq-like alignment policy also shifts the seed to the opposite end of the alignment. Thus, the reverse-complement cases will use the opposite index from the corresponding forward cases.

Recall that case-outer phased search (enabled via -z/--phased) examines cases in the outer loop and iterates through reads in the inner loop. This organization allows one index to be resident in memory at a time while minimizing the total number of forward/mirror index turnovers required during alignment. The introduction of the new reverse-complement cases potentially doubles the number of turnovers required in this mode. Bowtie avoids most of this cost by interleaving processing of the forward and reverse-complement cases, effectively "packing" them into a schedule requiring as few turnovers as possible. Figure 16 illustrates Bowtie's strategy for interleaving the forward and reverse-complement cases for Maq-like 2-mismatch search. Only one additional turnover is needed compared with the forward-only version. Other alignment policies are interleaved similarly.

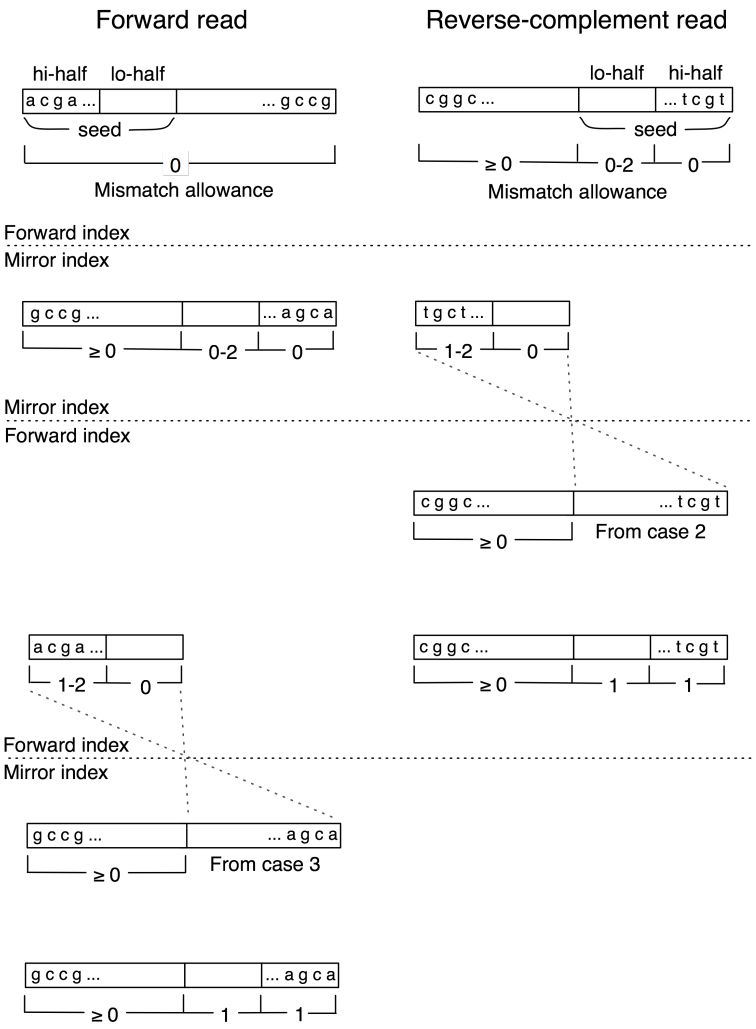

Figure 16 Case-outer phased search using both forward and reverse-complement read. The search requires only one additional index turnover compared to the search using only the forward read.

Strand bias

Strand bias occurs when (a) a read aligns equally well to several sites on both the forward and reverse-complement strands of the reference, and (b) the number of such sites on one strand differs from the number on the other strand. When this happens, Bowtie first chooses one strand or the other with 50% probability, then reports a random alignment for that read from among those possible on the selected strand. This tends to over-assign alignments to the sites

on the strand with fewer sites and under-assign to sites on the strand with more sites. The effect is mitigated, though it may not be eliminated, when reads are longer or when paired-end reads are used.

Bowtie counteracts strand bias by selecting a strand with a probability proportional to the number of sites on that strand. This requires knowing the number of equally good sites on each strand, which in turns requires calculating Burrows-Wheeler ranges for both strands. This is strictly more work than is required otherwise; an aligner that simply allows strand bias need only obtain one of the two ranges in order to report an alignment.

Bowtie's strand bias correction is active only in best-first mode (--best). Strand bias is still an outstanding issue in Bowtie's default, depth-first search mode.

Paired-end alignment

Paired-end reads comprise two separate reads ("mates") that are sequenced from either end of a longer sequence ("insert"). Mate and insert lengths depend on parameters of the library preparation process. Typical mate lengths range from 35 up to more than 75 bases, depending on the library. Typical insert sizes range from 250 bases up to more than 10,000 bases, also depending on the library. Within a given library, paired-end reads are typically of uniform length and insert sizes vary somewhat from pair to pair.

No search strategy presented so far is practical for aligning a paired-end read. Handling both mates and the gap between them as part of a single search process requires traversing an intractable number of long paths between the mates

in the worst case. A more tractable alternative is to align the mates separately and combine pairs of individual mate alignments into paired-end alignments according to the insert-length constraint. When the search process discovers new alignments for one mate, a higher-level process intervenes and attempts to pair the new alignments up with previously observed alignments for the opposite mate such that the insert-length constraint is satisfied. In practice, this alternative is still very slow. This is for two reasons. First, the process of finding pairs of alignments that satisfy the insert-length constraint is quadratic in the total number of alignments. This becomes onerous when both mates align to repetitive sequences in the reference. Second, resolving the insert-length constraint requires knowing the reference offset of both mates, which in turn requires several repeated invocations of the LF(r) function. This can become onerous when either mate aligns to a repetitive sequence in the reference.

A yet more tractable alternative is to align the mates separately and concurrently as before, but, when a new alignment is identified for one "anchor" mate (which could be either mate), to immediately scan the appropriate section of the reference genome for a valid alignment for the "opposite" mate. This is how Bowtie performs paired-end alignment. The section of the reference to scan is determined by the location of the anchor alignment together with the insert-length constraint. The reference section is scanned starting in the center and radiating out to the left and right; this is because, if multiple alignments for the opposite mate exist within the section, we prefer to report those that are closest to the mean insert length. The algorithm used to find alignments for the opposite mate is

similar in spirit to Maq's. We store a 2-bit-per-base representation of the mates's seed portion in a 64-bit word and "slide" the word window across the reference. At each window position, a 64-bit word is built from the corresponding reference substring and the two words are exclusive-or'ed to obtain a mismatch mask. If the number of non-0 bit-pairs in the mask exceeds the seed limit, we proceed to the next window position. If the limit is not exceeded, we attempt to extend the alignment through the non-seed portion. Using 64-bit arithmetic makes this process relatively efficient.

Mates that align to repetitive sequences in the reference still pose a problem, since many reference-scanning trials may be necessary to find a satisfied paired-end alignment (or to determine that there is none). The problem is mitigated when one of the mates aligns uniquely or almost uniquely. If one mate aligns a small number of times, it is beneficial to choose that mate as the anchor since we need only perform a small number of total trials before we find a satisfied alignment or prove that none exists. But if both mates align to repetitive regions of the genome, we are forced to perform at least as many trials as there are alignments for the less ambiguous mate. To address this, Bowtie imposes a limit on the number of trials performed for a given paired-end read before giving up and declaring the pair unalignable. This may lead to some alignable pairs being declared unalignable, but this should only occur when both ends of the pair align to repetitive sequences in the reference; such alignments are not typically useful to the user. The user can configure or disable this limit via the --pairtries

and -y/--tryhard arguments. The insert-length constraint is configurable via the -I and -X options.

Because Bowtie's paired-end alignment strategy requires scanning the reference, the reference must be stored in the Bowtie index and must be present in memory when paired-end reads are being aligned. This increases Bowtie's memory footprint by the size of the 2-bit-per-base-encoded reference sequence, which for the human genome contributes another 680 megabytes of disk space and memory. Bowtie's total memory footprint when aligning paired-end reads against the human genome is therefore about 2.9 gigabytes.

Results

Performance results were obtained using reads from the 1000 Genomes project pilot [NCBI Short Read Archive:SRR001115]. A total of 8.84 million reads, about 1 lane of data from an Illumina instrument, were trimmed to 35 bases and aligned to the human reference genome [NCBI build 36.3]. Unless specified otherwise, read data is not filtered or modified (besides trimming) from how it appears in the Archive. This leads to about 70-75% of reads aligning somewhere to the genome. This is typical for raw data from the Archive. More aggressive filtering leads to higher alignment rates and faster alignment.

All runs were performed on a single CPU. Bowtie speedups were calculated as a ratio of wall-clock alignment times. Both wall-clock and CPU times are given to demonstrate that I/O load and CPU contention are not significant factors.

The time required to build the Bowtie index was not included in the Bowtie running times. Prior to the alignment step, the user must compute an index for the reference genome, which can then be re-used across many alignment runs. We anticipate most users will simply download such indices from a public repository. The Bowtie website [31] provides indices for current builds of the human, chimp, mouse, dog, rat, and Arabidopsis thaliana genomes, among others.

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

Comparison to SOAP, Maq and BWA

Maq is a popular aligner [1, 4, 5, 32, 33] that is among the fastest competing open source tools for aligning millions of Illumina reads to the human genome. SOAP is another open source tool that has been published and used in short-read projects [6, 34]. Table 1 presents the performance and sensitivity of Bowtie v0.9.6, Bowtie v0.9.9.3 (for --best results), SOAP v1.10 and Maq v0.6.6. SOAP v1.10 could not be run on the PC because SOAP's memory footprint exceeds the PC's physical memory. The "soap.contig" version of the SOAP binary was used. For comparison with SOAP, Bowtie was invoked with "-v 2" to mimic SOAP's default matching policy (which allows up to 2 mismatches in the alignment and disregards quality values), and with "--maxns 5" to simulate SOAP's default policy of filtering out reads with 5 or more no-confidence bases.

For the Maq comparison Bowtie is run with its default policy, which mimics Maq's default policy of allowing up to 2 mismatches in the first 28 bases 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 the experiments where the "--best" option is not specified to ensure only the forward or mirror index is resident in memory at one time. To demonstrate Bowtie's performance when best-first search is used, we also show results using the latest version of Bowtie (0.9.9.3) with the "--best" option specified. For those runs, Bowtie is not invoked with "-z", since "-z" is incompatible with best-first search.

The number of reads aligned indicates that SOAP (67.3%) and Bowtie –v 2 (67.4%) have comparable sensitivity. 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 (without --best) (71.9%) also have roughly comparable sensitivity, though 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 some flexibility in Maq's alignment algorithm that allows some alignments to have 3 mismatches in the seed. The remainder of the reads mapped by Maq but not Bowtie are due to Bowtie's backtracking ceiling. Note that Bowtie --best makes up some (0.1 percentage points) of the gap by having a higher backtracking ceiling.

Maq's documentation mentions that reads containing "poly-A artifacts" can impair Maq's performance. Table 2 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 1, and the same observations about the relative sensitivity of Bowtie and Maq apply here.

Table 1 Performance and sensitivity of Bowtie v0.9.6, SOAP v1.10 and Maq v0.6.6 when aligning 8.84M reads from the 1000 Genome project [NCBI Short Read Archive:SRR001115] trimmed to 35 base pairs. The "soap.contig" version of the SOAP binary was used. SOAP could not be run on the PC because SOAP's memory footprint exceeds the PC's physical memory. For the SOAP comparison, Bowtie was invoked with "-v 2" to mimic SOAP's default matching policy (which allows up to 2 mismatches in the alignment and disregards quality values). For the Maq comparison Bowtie is run with its default policy, which mimics Maq's default policy of allowing up to 2 mismatches in the first 28 bases and enforcing an overall limit of 70 on the sum of the quality values at all mismatched positions. To make Bowtie's memory footprint more comparable to Maq's, Bowtie is invoked with the "-z" option in all experiments to ensure only the forward or mirror index is resident in memory at one time.

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (MB)	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
Bowtie ==best (server)	46m:54s	47m:23s	11.2	2,383	2.6x	72.0
Maq (server)	32h:56m:53s	32h:58m:39s	0.27	804	107x	74.7

Table 2 Performance and sensitivity of Bowtie v0.9.6 and Maq v0.6.6 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 1.

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (MB)	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
Bowtie --best (server)	42m:38m	43m:28s	12.2	2,383	2.3x	75.1
Maq (server)	18h:49m:07s	18h:50m:16s	0.47	804	60.2x	78.0

Read length and performance

As sequencing technology improves, read lengths are growing beyond the 30-50 bases commonly seen in public databases today. Bowtie, Maq, and SOAP support reads of lengths up to 1024, 63, and 60 bases respectively, and Maq versions 0.7.0 and later support read lengths up to 127 bases. Table 3 shows performance results when the three tools are each used to align three sets of 2M untrimmed reads, a 36-base set, a 50-base set and a 76-base set, to the human genome on the server platform. Each set of 2M is randomly sampled from a larger set [NCBI Short Read Archive: SRR003084 for 36-base, SRR003092 for 50-base, SRR003196 for 76-base]. Reads were sampled such that the three sets of 2M have uniform per-base error rate, as calculated from per-base 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 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-base set because v0.6.6 cannot align reads longer than 63 bases. SOAP was not run on the 76-base set because it does not support reads longer than 60 bases.

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- to 50-base reads but is substantially slower for 76-base reads, though it is still more than an order of magnitude faster than Maq.

Table 3 The performance of Bowtie v0.9.6, SOAP v1.10, Maq versions v0.6.6 and v0.7.1 on the server platform when aligning 2M untrimmed reads from the 1000 Genome project [NCBI Short Read Archive: SRR003084 for 36-base, SRR003092 for 50-base, SRR003196 for 76-base]. For each read length, the 2M reads were randomly sampled from the FASTQ file downloaded from the Archive such that the average per-base error rate as measured by quality values was uniform across the three sets. All reads pass through Maq's "catfilter". Maq v0.7.1 was used for the 76-base reads because v0.6.6 does not support reads longer than 63 bases. SOAP is excluded from the 76-base experiment because it does not support reads longer than 60 bases. Other experimental parameters are identical to those of the experiments in Table 1.

Length (bases)	Program	CPU time	Wall clock time	Peak virtual memory footprint (MB)	Bowtie speedup	Reads aligned (%)
36	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	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	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

Paired-end performance

Table 4 presents the performance and sensitivity of Bowtie v0.9.9.3 and Maq v0.6.6 when aligning 6 million human paired-end reads with mates of length 48 and 40. The paired-end reads are taken from the Short Read Archive (Run accession SRR001802). Bowtie is run both in "-v 2" mode, where neither mate is permitted to align with more than 2 mismatches, and in "-n 2" mode, which enforces the Maq-like alignment policy on both mates, where no more than 2 mismatches are permitted in the seed and the quality ceiling is 70. Maq's paired-end aligner uses an approach that, like Bowtie's, first aligns one mate as an anchor, then searches for the opposite mate by scanning the relevant region of the reference. If the opposite mate cannot be found in the initial pass, Maq makes a second attempt using Smith-Waterman [16] to scan the region, enforcing a looser alignment policy that allows gaps. Bowtie uses a sliding seed-and-extend window that enforces the same alignment policy as for the anchor mate. As shown in Table 4, Maq's more sensitive policy (both for the anchor and for the opposite mate) allows it to align more pairs than Bowtie.

Note that since Bowtie's paired-end alignment strategy requires scanning regions of the reference, the reference string must be present in memory. This leads to a larger memory footprint. As shown in Table 4, Bowtie's memory footprint is significantly larger than Maq's, though it is still small enough to be run on a workstation with 4 gigabytes of RAM.

Table 4 The performance of Bowtie v0.9.9.3 and Maq v0.6.6 on the server platform when aligning 6M untrimmed 48x40-base paired-end reads from the 1000 Genome project [NCBI Short Read Archive: SRR001802]. All reads pass through Maq's

"catfilter". Bowtie speedup for the Maq row is calculated with respect to Bowtie with its default options (not with –v 2, which is further from Maq's default). Other experimental parameters are identical to those of the experiments in Table 1.

	CPU time	Wall clock time	Pairs mapped per hour (millions)	Peak virtual memory footprint (MB)	Bowtie speedup	Pairs aligned (%)	Additional pairs SW aligned (%)
Bowtie –v 2	1h:32m:44s	1h:33m:21s	1.3	3,029	-	42.8	-
Bowtie	2h:14m:01s	2h:14m:56s	0.89	3,021	-	57.4	-
Maq	20h:24m:35s	20h:26m:07s	0.074	1,131	9.1x	65.6	4.1

Parallel performance

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 global bookkeeping, such as marking a read as "done." 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, so footprint does not increase substantially when multiple threads are used. Table 4 shows performance results for running Bowtie v0.9.6 on the 4-core server with 1, 2, and 4 threads.

Table 5 Performance results for running Bowtie v0.9.6 on the 4-core server with 1, 2, and 4 threads. Other experimental parameters are identical to those of the experiments in Table 1.

	CPU time	Wall clock time	Reads mapped per hour (millions)	Peak virtual memory footprint (MB)	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

Chapter 4: Time-space tradeoffs in Burrows-Wheeler indexing

This chapter explores Burrows-Wheeler index construction. First we introduce a recently discovered blockwise technique that makes it possible to construct the suffix array and Burrows-Wheeler index in a relatively small memory footprint. We measure some of the approach's tradeoffs and show that, for blockwise index construction, the difference cover sample appears to be a critical factor in achieving good performance for genomes as repetitive as the human genome. Finally, we present performance results achieved using the Bowtie index builder on the human genome.

Blockwise index construction

We previously visualized building the Burrows-Wheeler Transform as the process of (a) building the Burrows-Wheeler Matrix, i.e. the matrix whose rows are all cyclic rotations of the input text followed by $ sorted lexicographically, and then (b) reading the characters of BWT(T) from the last column of the matrix (Figure 1). But the relationship between the Burrows-Wheeler Matrix and the Suffix Array (SA) suggests an alternate way to define and build BWT(T) in terms of the text T and the suffix array SA:

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

Figure 17 Alternate definition of the Burrows-Wheeler Transform BWT(T) in terms of the original text T and the suffix array of the original text, SA.

There are well known ways of constructing the suffix array efficiently. However, for the human genome, the memory footprints of these techniques generally exceed the physical memory of a typical workstation. About 11-12 gigabytes of space is required simply to store the final answer. Since most techniques calculate the answer in memory, their footprint will be at least this large. Popular suffix-sorting algorithms like Larsson-Sadakane [35] and Manber-Myers [36] construct the suffix array and the inverse suffix array in tandem, incurring a peak memory footprint about twice the size of the suffix array: ~22-24 gigabytes for the human genome.

Recent work by Kärkkäinen [37] proposes a blockwise technique that builds the suffix array block-by-block, discarding each block after calculating the corresponding block of the BWT. This can reduce peak memory usage by obviating the need to store the entire suffix array in memory. The maximum size of a block is given by a parameter b_{max}. The overall blockwise BWT construction algorithm is as follows:

1. From the set of all suffixes, choose a random sample of "splitter" suffixes. Include the lexicographically greatest and least suffixes as splitters.
2. Sort the splitters lexicographically
3. For each consecutive pair of splitters, starting with lexicographically smallest, do:
    a. For each suffix of the input text, add it to list L if it falls lexicographically between the two splitters

b. Sort all suffixes in L, thus making L a contiguous block of the suffix array

c. Compute the BWT block corresponding to L and discard L

4. Output the concatenation of all BWT blocks computed in 3c

Care must be taken to choose splitters that yield a relatively even distribution of sizes for the suffix array blocks. Bowtie's implementation uses something akin to the heuristic suggested by Kärkkäinen [37] of choosing more splitters than are needed, determining the sizes of the resulting buckets using Manber-and-Myers binary search [36], then doing a round of splitting and merging and, if any blocks have more than b_{max} elements, iterating the process again.

Kärkkäinen's goal is an algorithm with sub-quadratic time and sub-linear space complexity. The potentially problematic steps are 2, 3a, and 3b. Kärkkäinen proposes an algorithm based on Z boxes for step 3a that is linear in the length of the input text. Two problems remain. First, the Z-box-based algorithm requires space linear in the length of the text in order to store the Z values. Second, neither of the suffix sorting steps (2 and 3b) can be performed straightforwardly in both sub-quadratic time and sublinear space.

Kärkkäinen's solution to both problems is the difference cover (DC) sample, a concept first applied to suffix sorting by Burkhardt and Kärkkäinen [38]. The idea is to first sort a relatively small subset (sample) of the suffixes, then use the lexicographical ordering of the samples to help break difficult "ties"

in later stages. A key parameter is the "period" of the difference cover sample, *v*, which dictates the density of the sample, and also ultimately dictates the worst-case amount of work that must be done to determine the relative lexicographical ordering of two strings given the sample. With the help of the sample, the sorting steps (2 and 3b) run in O(n log n + *v*n) time, and the binary-search step (3a) runs in O(v) space. The sample itself can be calculated in O(n log n + v) time [38].

Effect of block size on indexing performance

Table 5 shows the impact of maximum block size (b_{max}) on the time and memory performance of an early version of the Bowtie blockwise indexer. Results are shown for three texts of different sizes: human chromosomes 21 (25 Mbases) 11 (131 Mbases) and 1 (226 Mbases). All DNA sequences used were downloaded from the contig version of NCBI human genome build 36.3. Experiments were performed on a server with 32GB of physical RAM and 4 dual-core 2.2 GHz AMD Opteron 875 Processors.

Table 6 Impact of maximum block size on the time and memory performance of the Bowtie blockwise indexer. The difference cover period is fixed at 1024 for all runs.

Run Description	Wall clock running time	Peak virtual memory footprint
Chromosome 21 (25 Mbases)		
Max block size: 32M (2 blocks)	0m:51s	329 MB
Max block size: 16M (3 blocks)	0m:56s	192 MB
Max block size: 8M (6 blocks)	0m:58s	124 MB
Max block size: 4M (13 blocks)	1m:19s	Not measured
Max block size: 2M (25 blocks)	1m:47s	Not measured
Max block size: 1M (49 blocks)	2m:51s	Not measured

Chromosome 11 (131 Mbases)		
Max block size: 128M (2 blocks)	3m:39s	977 MB
Max block size: 64M (4 blocks)	4m:18s	562 MB
Max block size: 32M (5 blocks)	3m:56s	381 MB
Max block size: 16M (10 blocks)	5m:55s	264 MB
Max block size: 8M (22 blocks)	7m:11s	206 MB
Max block size: 4M (44 blocks)	9m:54s	185 MB
Chromosome 1 (226 Mbases)		
Max block size: 128M (3 blocks)	10m:07s	1,155 MB
Max block size: 64M (5 blocks)	11m:17s	870 MB
Max block size: 32M (9 blocks)	11m:26s	543 MB
Max block size: 16M (19 blocks)	13m:28s	434 MB
Max block size: 8M (38 blocks)	18m:13s	387 MB
Max block size: 4M (77 blocks)	27m:25s	374 MB

The results exhibit a consistent, direct relation between the number of blocks and the running time of the algorithm. This is expected, since the construction of each block requires a separate pass over the entire input text (step 3a). There is also an inverse relation between the number of blocks and memory usage. This too is expected since fewer and larger blocks lead to more peak and average memory usage for storing and sorting those blocks.

For each chromosome, there seems to be a point of diminishing returns in the time/space tradeoff as the number of blocks increases. Beyond that point, peak and average memory usage decrease very little while runtime increases substantially. This implies that for a given input text there will generally be a "sweet spot" where the blockwise construction yields a substantial space savings

without affecting runtime too badly. In these experiments, that point lies at roughly 10 blocks.

The number of blocks for a particular run does not correspond cleanly to the maximum block size (e.g. halving the block size from 64M to 32M leads to only 1 additional block for Chromosome 11). This is owing to randomness in the process of selecting samples, as well as the tendency of the splitter heuristic to settle quickly on a "good enough" set of bucket boundaries.

Effect of difference cover and period on indexing performance

The difference cover sample breaks lexicographical "ties" between two suffixes in constant time when those suffixes share a sufficiently long prefix. In Kärkkäinen's algorithm, the difference cover sample is constructed up front and then used to reduce the asymptotic complexity of sorting (in steps 2 and 3b) and block accumulation (step 3a). Like Kärkkäinen's, Bowtie's implementation constrains the period to be a power of 2 to avoid costly division and modulus operations. Table 6 shows how disabling the difference cover sample and adjusting its period affects runtime and memory usage for experiments using chromosomes 1, 11 and 21.

Table 7 Impact of difference-cover period on the time- and memory-performance of the Bowtie blockwise indexer. The maximum block sizes are set to 16 M for all Chromosome 21 runs, 32 M for all Chromosome 11 runs, and 64M for all Chromosome 1 runs.

Run Description	Wall clock running time	Peak virtual memory footprint
Chromosome 21 (25 Mbases)		
No difference cover	1m:00s	154 MB
Difference cover period = 4096	1m:01s	176 MB

Difference cover period = 2048	1m:01s	157 MB
Difference cover period = 1024	0m:56s	192 MB
Difference cover period = 512	1m:00s	168 MB
Difference cover period = 256	0m:51s	193 MB
Chromosome 11 (131 Mbases)		
No difference cover	4m:32s	337 MB
Difference cover period = 4096	4m:30s	338 MB
Difference cover period = 2048	5m:18s	348 MB
Difference cover period = 1024	3m:56s	375 MB
Difference cover period = 512	4m:46s	379 MB
Difference cover period = 256	4m:34s	386 MB
Chromosome 1 (226 Mbases)		
No difference cover	23m:00s	830 MB
Difference cover period = 4096	16m:29s	867 MB
Difference cover period = 2048	12m:27s	836 MB
Difference cover period = 1024	11m:17s	870 MB
Difference cover period = 512	10m:47s	833 MB
Difference cover period = 256	9m:10s	980 MB

While the difference cover does not significantly affect the time or memory performance of indexing Chromosomes 21 or 11, it substantially improves the running time for indexing Chromosome 1. As expected, the improvement grows as the period decreases. That the effect is observed only for Chromosome 1 is likely owing to Chromosome 1's repeat content. In no experiment did the construction and use of the difference cover have an obvious effect on memory

footprint, though such an effect would likely be visible for periodicities less than 512.

Use of difference cover seems to be critical in keeping the runtime of highly repetitive sequences under control, and a periodicity in the neighborhood of 512 seems to be a reasonable trade between added memory overhead and runtime improvement.

Performance for human genome

Table 7 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. "2-bit-per-base references" indicates whether a bit-packed representation of the reference sequence was used. The bit-packed representation reduces memory footprint but increases running time.

Table 8 Memory footprints and wall clock times for a human-genome run of the indexer under parameters selected to satisfy different physical memory constraints. 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. "2-bit-per-base references" indicates whether a bit-packed representation of the reference sequence was used.

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

Chapter 5: Improved scalability and convenience with Cloud Computing

This chapter presents the design of Crossbow, an adaptation of Bowtie to a Cloud Computing context. Performance results are reported showing that Crossbow is capable of aligning about 14.3x coverage of human reads in about 1 hour and 11 minutes of wall-clock time.

Introduction

With the advent of robust implementations of cloud computing software and services such as Hadoop [20] and Amazon Web Services [21], it is increasingly possible to solve very data-intensive problems efficiently without ever buying or maintaining sophisticated computer equipment. This chapter presents a framework called Crossbow that combines the speed advantage of Bowtie with the computational capacity available from Amazon's Elastic Compute Cloud (EC2) service to align reads and detect single-nucleotide variations in human datasets.

Crossbow combines the Bowtie aligner with a simple single-nucleotide variant detector. Both the aligner and the variant detector have been customized to work together in a MapReduce [22] context. Crossbow runs in any environment that supports the Hadoop [20] MapReduce implementation, including on Hadoop "instances" (virtual machines) provided by Amazon for use within the Elastic Compute Cloud (EC2) web service. We present results demonstrating that Crossbow is capable of aligning about 14.3x-coverage worth

of human Illumina reads in 1 hour and 11 minutes using an EC2 cluster of 20 Extra-Large High-CPU nodes (1 master, 19 slaves), incurring a total of about $32 in cluster rental fees. Since Amazon EC2 is available to anyone with an Amazon AWS account, and because our work uses publicly available EC2 machine images in combination with scripts that can be downloaded from the Bowtie website, our technique and results are readily reproducible by others.

Variation detection in MapReduce

The insight behind Crossbow is that variation detection problem can be factored into a Map function: alignment, and a Reduce function: variant detection over a contiguous stretch of the reference. Crossbow's Map function takes as input a key/value pair representing a single read with quality values (in FASTQ format) and aligns the read to the human genome using the Bowtie alignment algorithm with its default arguments, which reports at most one alignment per read or read pair. The output of the Map function is empty if Bowtie reports no alignments. If Bowtie reports an alignment, the output is a key/value pair where the key is an identifier that uniquely identifies the contiguous stretch of the reference text that was aligned to (e.g., offsets 100,000-200,000 of human Chromosome 1), and the value is the alignment itself. Crossbow's Reduce function takes a bundle of key/value pairs output by the Map function where all bundled pairs share a particular key (indicating that all bundled alignments are located along the same contiguous stretch of the reference genome). The Reduce function then sorts the bundled alignments along the length of the reference, forming a multiple alignment including the reads and the reference, then examines

each column of the multiple alignment and detects variations using a simple model that scores the significance of a hypothetical homozygous single-nucleotide variation. The output of the Reduce function is a (possibly empty) list of key/value pairs where each pair describes a single detected variant.

Organizing Crossbow as Map and Reduce functions enables it to run in the context of a MapReduce implementation such as Hadoop. Hadoop applications can be readily scaled to a very large number of computers. The Hadoop infrastructure abstracts away a number of the engineering questions about, for example, how to recruit a set of computers into a cluster, how to spawn Map and Reduce tasks on those computers, how to forward the input key/value pairs to the mappers, how to bundle the output key/value pairs output by the mappers, how to kill the mappers, how to spawn the reducers, how to forward the bundled mapper outputs to the appropriate reducers, and how to collect and store the output key/value pairs output by the reducers.

Another advantage is that Hadoop applications can be run "in the cloud" using Amazon's EC2 service. Any researcher with an Amazon AWS account and sufficient funds can run large-scale applications on EC2. Because the EC2 infrastructure is generic and not tied to vagaries of a particular cluster setup, results obtained in EC2 are easy for other researchers to recreate.

Simple Storage Service

Datasets provided as input to Crossbow are typically quite large. For the experiment described in this paper, the input consists of about 130 GB of uncompressed FASTQ files. We first upload the input data to Amazon's Simple

Storage Service (S3). Using 10 EC2 nodes to simultaneously download the input data from the Short Read Archive and upload it into S3 takes about 3-5 hours. During this time, fees are assessed for use of the nodes (about $1 per hour) and for the amount of data transferred (about $13 for this dataset). Once the transfer is complete, a monthly fee is assessed per gigabyte of data stored in S3 (about $20 per month for this dataset). If the time delay and/or fees are undesirable the user, the user always has the option of running Crossbow on a local Hadoop cluster where the data is available locally.

Adapting Bowtie to Hadoop

Several minor customizations were made to the Bowtie aligner to allow it to operate within Hadoop. These included (a) adding an option to take read input from standard-in, (b) changing the output to optionally include a "partition key," which acts as the key for pairs emitted by the mapper.

Crossbow's Reduce function takes a bundle of all key/value pairs having a particular key, so the Bowtie aligner had to be modified to output this key. The key consists of a chromosome identifier followed by an ASCII space character followed by the offset of the alignment within the chromosome divided by the partition size. The partition size is simply an upper bound on the length of genome that may be represented by a particular key. Crossbow sets the partition size to an appropriate value (the experiments described in this thesis used a value of 100,000) via a command-line option when it invokes the Bowtie aligner.

A problem arises when an alignment generated by the Bowtie aligner spans multiple partitions. Assigning the read to only one or the other of the two

partitions creates an artificial "breakpoint" in the overall multiple alignment at the junction and deprives the variant detector of some relevant information. This in turn could cause spurious mispredictions in the reducer. Thus, Bowtie generates a separate alignment report (with the same value but a different key) for each partition overlapped by a particular alignment.

Results

To test Crossbow's ability to detect variations with respect to a large dataset, an experiment was performed using 20 High-CPU Extra Large Instances (1 master, 19 slaves) from Amazon's EC2 service; each instance has 7 GB RAM and 8 cores, each approximately 3.0 Ghz. The input data consisted of 129 GB of FASTQ reads from the 1000 Genomes pilot project, obtained from the NCBI Short Read Archive; this constitutes 14.3x coverage of the human genome (before alignment). The input data, binaries for the Bowtie aligner and variant caller, and a Bowtie index of the human genome (NCBI Build 36.3, assembled) were all initially uploaded to S3. 12 mappers were run on each instance and 12 reducers were run in total. Output was written directly to a directory in S3.

At 80¢ per node per hour, the experiment cost $16 per hour for all 20 nodes. 12 mappers were run per instance even though each instance has only 8 cores in an attempt to oversubscribe each instance to better hide the latency of fetching data from S3 and the idleness that co-occurs with task switchovers.

The experiment required 1 hour and 11 minutes to run. The overall experiment detected about 727,000 single-base substitutions. This number is lower than the expected total number of SNPs per individual, which is

approximately 3-4 million. This is not surprising since (a) only about 60-70% of the reads aligned, (b) coverage of the reads that do align is not evenly distributed over the whole genome, and (c) the variation detector only detects homozygous single-base substitutions, whereas many human substitutions are observed to be heterozygous. Extending the Crossbow SNP caller to detect heterozygous SNPs is future work.

Bibliography

1. Down TA, Rakyan VK, Turner DJ, Flicek P, Li H, Kulesha E, Graf S, Johnson N, Herrero J, Tomazou EM, Thorne NP, Backdahl L, Herberth M, Howe KL, Jackson DK, Miretti MM, Marioni JC, Birney E, Hubbard TJ, Durbin R, Tavare S, Beck S: **A Bayesian deconvolution strategy for immunoprecipitation-based DNA methylome analysis**. *Nat Biotechnol* 2008, **26**:779-785.
2. Johnson DS, Mortazavi A, Myers RM, Wold B: **Genome-wide mapping of in vivo protein-DNA interactions**. *Science* 2007, **316**:1497-1502.
3. Marioni JC, Mason CE, Mane SM, Stephens M, Gilad Y: **RNA-seq: An assessment of technical reproducibility and comparison with gene expression arrays**. *Genome Res* 2008.
4. Bentley DR, Balasubramanian S, Swerdlow HP, Smith GP, Milton J, Brown CG, Hall KP, Evers DJ, Barnes CL, Bignell HR, Boutell JM, Bryant J, Carter RJ, Keira Cheetham R, Cox AJ, Ellis DJ, Flatbush MR, Gormley NA, Humphray SJ, Irving LJ, Karbelashvili MS, Kirk SM, Li H, Liu X, Maisinger KS, Murray LJ, Obradovic B, Ost T, Parkinson ML, Pratt MR *et al*: **Accurate whole human genome sequencing using reversible terminator chemistry**. *Nature* 2008, **456**:53-59.
5. Ley TJ, Mardis ER, Ding L, Fulton B, McLellan MD, Chen K, Dooling D, Dunford-Shore BH, McGrath S, Hickenbotham M, Cook L, Abbott R, Larson DE, Koboldt DC, Pohl C, Smith S, Hawkins A, Abbott S, Locke D, Hillier LW, Miner T, Fulton L, Magrini V, Wylie T, Glasscock J, Conyers J, Sander N, Shi X, Osborne JR, Minx P *et al*: **DNA sequencing of a cytogenetically normal acute myeloid leukaemia genome**. *Nature* 2008, **456**:66-72.
6. Wang J, Wang W, Li R, Li Y, Tian G, Goodman L, Fan W, Zhang J, Li J, Zhang J, Guo Y, Feng B, Li H, Lu Y, Fang X, Liang H, Du Z, Li D, Zhao Y, Hu Y, Yang Z, Zheng H, Hellmann I, Inouye M, Pool J, Yi X, Zhao J, Duan J, Zhou Y, Qin J *et al*: **The diploid genome sequence of an Asian individual**. *Nature* 2008, **456**:60-65.
7. Li H, Ruan J, Durbin R: **Mapping short DNA sequencing reads and calling variants using mapping quality scores**. *Genome Res* 2008.
8. Li R, Li Y, Kristiansen K, Wang J: **SOAP: short oligonucleotide alignment program**. *Bioinformatics* 2008, **24**:713-714.
9. Kaiser J: **DNA sequencing. A plan to capture human diversity in 1000 genomes**. *Science* 2008, **319**:395.
10. Smith AD, Xuan Z, Zhang MQ: **Using quality scores and longer reads improves accuracy of Solexa read mapping**. *BMC Bioinformatics* 2008, **9**:128.
11. Lin H, Zhang Z, Zhang MQ, Ma B, Li M: **ZOOM! Zillions Of Oligos Mapped**. *Bioinformatics* 2008.

12. **SHRiMP - SHort Read Mapping Package**. http://compbiocstorontoedu/shrimp/.
13. Baeza-Yates RA, Perleberg CH: **Fast and practical approximate string matching**. *Inf Process Lett* 1996, **59**:21-27.
14. Burkhardt S, Kärkkäinen J: **Better Filtering with Gapped q-Grams**. *Fundam Inf* 2003, **56**:51-70.
15. Ma B, Tromp J, Li M: **PatternHunter: faster and more sensitive homology search**. *Bioinformatics* 2002, **18**:440-445.
16. Smith TF, Waterman MS: **Identification of common molecular subsequences**. *J Mol Biol* 1981, **147**:195-197.
17. Ferragina P, Manzini G: **Opportunistic data structures with applications**. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science.* IEEE Computer Society; 2000.
18. Ferragina P, Manzini G: **An experimental study of an opportunistic index**. In: *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms.* Washington, D.C., United States: Society for Industrial and Applied Mathematics; 2001.
19. 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**:R25.
20. **Welcome to Apache Hadoop Core!** http://hadoopapacheorg.
21. **Amazon Web Services**. http://awsamazoncom/.
22. Dean JaG, S.: **MapReduce: simplified data processing on large clusters**. *Commun ACM* 2008, **51**.
23. Healy J, Thomas EE, Schwartz JT, Wigler M: **Annotating large genomes with exact word matches**. *Genome Res* 2003, **13**:2306-2315.
24. Lippert RA: **Space-efficient whole genome comparisons with Burrows-Wheeler transforms**. *J Comput Biol* 2005, **12**:407-415.
25. 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**:i195-204.
26. Lam TW, Sung WK, Tam SL, Wong CK, Yiu SM: **Compressed indexing and local alignment of DNA**. *Bioinformatics* 2008, **24**:791-797.
27. Burrows M, Wheeler DJ: **A block sorting lossless data compression algorithm**. *Digital Equipment Corporation, Palo Alto, CA* 1994, **Technical Report 124**.
28. Kurtz S: **Reducing the Space Requirement of Suffix Trees**. *Software – Practice and Experience* 1998, **29**:1149 - 1171.
29. Ewing B, Green P: **Base-calling of automated sequencer traces using phred. II. Error probabilities**. *Genome Res* 1998, **8**:186-194.
30. Herold J, Kurtz S, Giegerich R: **Efficient computation of absent words in genomic sequences**. *BMC Bioinformatics* 2008, **9**:167.
31. **Bowtie: An ultrafast, memory-efficient short read aligner**. http://bowtie-biosourceforgenet/indexshtml.
32. Campbell PJ, Stephens PJ, Pleasance ED, O'Meara S, Li H, Santarius T, Stebbings LA, Leroy C, Edkins S, Hardy C, Teague JW, Menzies A,

Goodhead I, Turner DJ, Clee CM, Quail MA, Cox A, Brown C, Durbin R, Hurles ME, Edwards PA, Bignell GR, Stratton MR, Futreal PA: **Identification of somatically acquired rearrangements in cancer using genome-wide massively parallel paired-end sequencing**. *Nat Genet* 2008, **40**:722-729.

33. Holt KE, Parkhill J, Mazzoni CJ, Roumagnac P, Weill FX, Goodhead I, Rance R, Baker S, Maskell DJ, Wain J, Dolecek C, Achtman M, Dougan G: **High-throughput sequencing provides insights into genome variation and evolution in Salmonella Typhi**. *Nat Genet* 2008, **40**:987-993.
34. Nagalakshmi U, Wang Z, Waern K, Shou C, Raha D, Gerstein M, Snyder M: **The transcriptional landscape of the yeast genome defined by RNA sequencing**. *Science* 2008, **320**:1344-1349.
35. Larsson N, Sadakane, K.: **Faster Suffix Sorting**. *Theoretical Computer Science* 2007, **387**:258 - 272.
36. Manber U, Myers G: **Suffix arrays: a new method for on-line string searches**. In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms.* San Francisco, California, United States: Society for Industrial and Applied Mathematics; 1990.
37. Kärkkäinen J: **Fast BWT in small space by blockwise suffix sorting**. *Theor Comput Sci* 2007, **387**:249-257.
38. Burkhardt S, Kärkkäinen J: **Fast lightweight suffix array construction and checking**. In: *14th Annual Symposium on Combinatorial Pattern Matching: 2003*; 2003.