Performance of genomics data analysis applications and pipelines in computational systems

Informe sobre el rendimiento de las aplicaciones y pipelines de análisis en sistemas de computación

Designing RISC-V-based Accelerators for next generation Computers (DRAC)

Código de Proyecto: 001-P-001723

Número de entregable:	E3.1
Nombre de entregable:	Informe sobre el rendimiento de las aplicaciones y pipelines de análisis en sistemas de computación
Periodo cubierto:	mes 01 a mes 12
Revisión:	01
Fecha límite del entregable:	31 diciembre 2020
Fecha de entrega:	31 diciembre 2020
Fecha de inicio del proyecto:	1 junio 2019
Duración:	36 meses

Miembro responsable del proyecto: Universitat Autònoma de Barcelona

Autores del entregable: Antonio Espinosa, Santiago Marco, Juan Carlos Moure, David Castells, Quim Aguado

Este proyecto está financiado por el Fondo Europeo de Desarrollo Regional de la Unión Europea en el marco del programa Operativo FEDER de Cataluña 2014-2020 con una financiación de 2.000.000€ y con el soporte de la Secretaria de Universidades e Investigación

Grado de divulgación		
PU	Público	X
CO	Confidencial, solo para miembros del consorcio	

Generalitat de Catalunya
Departament d'Economia i Coneixement
Secretaria d'Universitats i Recerca

Unió Europea
Fons Europeu de Desenvolupament Regional

Historia del documento

Versión	Fecha	Descripción/Cambios	Razones
01	30/12/20	Primera versión documento	

1. Genomic sequencing and bioinformatics processing pipelines 5
1.1. Approximate string matching for genomic sequences 6
1.2. FM-index elements 8
1.3. Sequence alignment 10
1.3.1. Error model 11
1.3.2. Alignment classification 12
1.4. Approximate String Matching 13
1.4.1. Dynamic programming pair-wise alignment 14
1.4.2. Filtering algorithms 15
1.4.3. Exact filters. Candidate verification techniques 19
1.4.4. Approximate filters 20
2. Architectural design considerations 23
2.1. FM-index architectural design 23
2.1.1. FM-index memory access pattern 23
2.1.2. FM-index computational needs 24
2.1.3. FM-index operators 25
2.2. Sequence filtering accelerating computations 27
2.2.1. Block-wise computations 27
2.2.2. Bit-parallel algorithms 27
2.2.3. SIMD algorithms 28
3. Hardware support for bioinformatics applications 29
4. References 30

Executive summary

This report presents the requirements of WP3 objectives. High Throughput Sequencing (HTS) alignment algorithms have to guarantee well defined, quantifiable and reproducible accuracy. Mapping is a crucial stage in most resequencing pipelines and many other HTS analysis workflows. Good quality results generated by a mapper will heavily influence downstream analysis, ultimately leading to meaningful and accurate scientific findings in biological research and medical diagnostics.

HTS calls for high performance algorithms that can cope with increasing yields in the production of genomic data. Even if a mapping tool delivers high quality results, they could be of no use if the time spent computing them is impractical. Therefore, sequence alignment algorithms have to keep up with the pace of HTS technologies. The objective of this project is to improve some relevant high performance algorithms for sequence alignment that can be used in new acceleration computer architectures

1. Genomic sequencing and bioinformatics processing pipelines

The recent development of high throughput sequencing technologies (HTS) from Roche, Illumina, IonTorrent, Nanopore and Pacific bioscience companies has allowed the possibility of fast and affordable sequencing of any living organism. As the result of this technology, a single sequencer system produces millions of short sequences for each individual as a representation of the individual genome split in millions of short nucleotide sequences.

From the point of view of data processing, scientists have established different protocols that describe how these sequences must be processed to build an individual genome first and then search for specific scientific analysis considering the differences in the individual data compared with reference genomes [Marx 13].

The quality of the results obtained becomes very relevant as downstream analysis depends on the accuracy of the previous steps. A poor methodology or wrong choice of protocols may produce errors that affect the whole analysis. The current state of the art is the development of large and complex analysis pipelines [Hwang 15] where many stages are still being into analysis to find better performance, accuracy and quality-control mechanisms. Also, standardization and flexibility have become paramount in this field for every research effort [Gargis 15]

Challenges to be solved involve the processing of the data and the quality assurance of the results obtained. As modern sequencing technologies become more affordable and produce better quality results, the stress is being put in the computational analysis of the datasets. The scientific community needs strong good quality methods, algorithms, tools and pipelines that make a good use of computational resources to improve the current bottleneck of genomic data analysis.

High-throughput sequencing (HTS) principally denotes those technologies capable of sequencing in the order of millions of reads per run, relatively quickly (from a few days to several hours) at very low price (less than $1 per million bases). Using them, we can deep-sequence a genome (cover every base of a genome with a large number of reads) producing massive amounts of genomic data. These techniques are often called massive parallel DNA sequencing as they rely upon millions of reactions run simultaneously to achieve very high yields of production [Reuter 15], [Loman 12]

HTS alignment algorithms have to guarantee well defined, quantifiable and reproducible accuracy. Mapping is a crucial stage in most resequencing pipelines and many other HTS analysis workflows. Good quality results generated by a mapper will heavily influence downstream analysis, ultimately leading to meaningful and accurate scientific findings in biological research and medical diagnostics.

HTS calls for high performance algorithms that can cope with increasing yields in the production of genomic data. Even if a mapping tool delivers high quality results, they could be of no use if the time spent computing them is impractical. Therefore, sequence alignment algorithms have to keep up with the pace of HTS technologies. They have also led the development of many new analytical tools for measuring important biological processes (e.g. variant calling, splicing, detecting protein binding, gene annotation and expression, copy number variation, and more).

The objective of this project is to improve some relevant high-performance algorithms for sequence alignment that can be used in new acceleration computer architectures.

As bioinformatic applications evolve, their scope is increasing and becoming ever more complex. For that reason, applications cannot be fully understood without context and must be studied in the general view of large analysis pipelines involving very different components. Current HTS data analysis pipelines consist of several steps and involve numerous tools. Generally, these analysis pipelines are divided into three stages: primary, secondary, and tertiary analysis.

Primary analysis consists of machine specific steps to call base pairs and compute quality scores. Basically, the base calling step converts raw sequencing signals from the sequencer into bases (i.e. A,C,G,T, and N for unknown calls). Additionally, this stage is expected to deliver a quality metric of the calling step. For this, quality values are assigned to each base call to estimate the likelihood of an erroneous call at that base.

Secondary analysis pursues the reconstruction of the original genome using the reads. This can be achieved by means of or de-novo assembly techniques resulting a list of sequences aligned against a reference genome (read mapping).

Finally, tertiary analysis main aim is to give interpretation of the results produced by the secondary analysis. Sometimes this analysis integrates data coming from multiple experiments and samples. Examples of this analysis are gene annotation, differential expression studies, alternative splicing studies, detection of rare variants, association studies etc.

1.1. Approximate string matching for genomic sequences

Full-text self-indexes have become very common because of the latest developments in HTS technologies. Essentially, these data structures address the challenge of indexing genome-scale references on computers reducing the need of memory. Full-text indexes can even be coupled with compression algorithms in order to generate compressed full-text self-indexes, thus reducing memory requirements even further [Navarro 07] [Grossi 03] [Ferragina 09] [Siren 08].

From the list of common data structures to store full-text self-indexes the most relevant for genomics data processing are succinct data structures. These indexes aim to perform fast searches over massive amounts of data stored in little space.

The first structure of relevance is the *suffix array* $SA_T[0..n-1]$ of a text T. It is an array containing all the starting positions of the suffixes of T sorted in lexicographical order.

G	A	T	T	A	C	A	$
0	1	2	3	4	5	6	7

Suffix Array	Sorted Suffixes
SA[0]=7	$
SA[1]=6	A$
SA[2]=4	ACA$
SA[3]=1	ATTACA$
SA[4]=5	CA$
SA[5]=0	GATTACA$
SA[6]=3	TACA$
SA[7]=2	TTACA$

Figure 1-1: suffix array of the text T=GATTACA$, where $ denotes the end of the text. The character "$" is smaller than any other symbol belonging to the alphabet

Note that any substring P of the text T is at the same time a prefix of a text suffix. In this way, any pattern P can be binary searched over the suffixes of SA in *O(m log(n))* time as each step in the binary search requires comparing up to *m* symbols between the pattern P and the corresponding text suffix.

The Burrows Wheeler Transform (BWT) [Burrows 94] is a fundamental concept in the field of succinct and compressed full-text indexes. The BWT is a permutation of the original text which has useful properties when one has to implement not only compression, but also string searching. BWT arranges together characters followed by the same context. In most cases, this leads to runs of the same characters which makes the BWT permutation susceptible to being easily compressed.

Beyond the compression features, the BWT transform leads to a transformed text that depicts useful properties for string searching. To explain those properties we need to define the rank function *rank(c,i)* and the accumulated counters *C[a]*.

- rank(c,i) is defined as the occurrence function over the suffix of T_{bwt} from 0 to i.

  rank(c,i) = occ(c, T_{bwt}[0..i])

- C[a], where a ∈ Σ, is an array containing the number of characters in text T that are smaller than a.

  $$C[a] = \sum_{c<a} \text{occ}(c, T), \qquad \forall a \in T$$

The FM-index [Ferragina 00] exploits the BWT transform to build compressed text indexes. In the end, they figured out a way of exactly searching patterns using the BWT text and some auxiliary data structures. At the same time, they proposed compressing the BWT text so as to reduce the index space requirements to O(n) − bits, which in practice can index the 3GB of the human genome in less than 1GB depending on the compression algorithms. Despite their compression properties, most current FM-Index designs do not compress the BWT for the sake of providing fast query accesses as the overall memory usage without compression is quite reasonable in practice – about the same size of the text encoded in ASCII.

LF_c function

To search for an exact pattern, the FM-Index uses two sentinels to delimit an interval containing all the suffixes that have the pattern as a common prefix. The main idea behind the FM-index is to use this LF function to backward search a pattern by iteratively updating two sentinels and exploiting the properties of the BTW.

```
input: P pattern, T_BWT, C[]
output: SA_T interval for P
begin
      lo = 0
      hi = n - 1
      for j = m - 1 to 0 do
            c = P[j]
            lo = C[c] + rank(c, lo -1) + 1
            hi = C[c] + rank(c, hi)
      end
      return(lo, hi)
end
```

In this manner, the backward search delimits the SA_T rows that exactly match a given pattern in O(m) time. Moreover, we can compute the number of matching positions by subtracting the sentinels at any point of the search (i.e. occ(P, T) = hi − lo), without computing the actual matching positions in the text. In case lo > hi, we know that there is no exact match for the given pattern.

Note that once the SA interval for a given pattern is known, all the positions inside its range are encoded in SA-space. That means that, for a given i-th row the MT matrix we lack the corresponding position in the text (i.e. $SA_T[i]$). In order to decode positions from SA-Space to Text-Space we need to unwind the BWT until a known corresponding position is reached. In a trivial case, we would traverse the text backwards, applying the LF function s times, until the first row is reached (i.e. beginning of the text T). At that point we would know that the source i-th row corresponds to s suffix of the text.

1.2. FM-index elements

The FM-index consists of the BWT of the text and some auxiliary data structures used to accelerate the computation of the LFc function, decode positions from SA-space to Text-space and fetch substrings of the original text.

The first structure to consider is the C[] array used to accelerate the computation of the LFc function. This C[] array C[a ∈ Σ] stores the accumulated occurrences of each character in the text. In addition, the FM-index stores partial counters interleaved with the BWT text to accelerate the computation of the rank function. In this way, the BWT text is divided into equal blocks of b characters (b is the block length).

Each FM-Index block b_i, as in figure 1-2, contains b characters of the BWT (T_{bwt}^i) and the partial counters (c_i [a ∈ Σ]) holding the count for each character until that position.

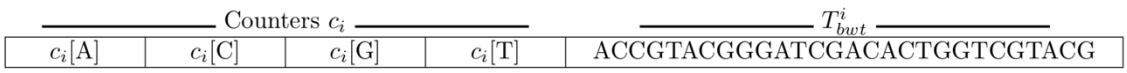

Figure 1-2: FM-index block design

The number of blocks in the FM-Index is equal to $|b_0,...,b_n| = \lceil |T|/b \rceil$ and that, for a given character *a*, the content of the partial counters is equal to
$c_i[a] = rank(a, i \cdot b) = occ(a, T_{bwt}[0..i \cdot b])$.

In this manner, computing *rank()* avoids counting occurrences up to this point and leaves the counting to potentially few characters past the counters. Depending on the FM-Index layout, regular inclusion of counters in the index will speed up rank queries at the cost of more index space. This is a trade-off between space consumption and cost per LF-operation.

Additionally, so as to accelerate the decoding of positions from SA-space to Text-space, any efficient FM-Index implementation stores samples of the SA. Depending on which sampling scheme is chosen, the average number of LF operations performed to reach a sampled position will vary. Also, different schemes allow for better average performances at the expense of extra index space.

For many indexed approximate string matching algorithms, it is also very important to be able to retrieve any substring of the original reference text efficiently. Note that most alignment algorithms end up pairwise aligning the pattern against several candidate text regions. For that, efficient recovery of text regions is of key relevance. However, as it is initially formulated, the FM-Index can only retrieve text regions by means of iterating LF operations in $O(|P|)$ time. Not only does this text retrieval mechanism perform several sparse index accesses before retrieving the whole text region, but each LF operation also involves several instructions just to retrieve a single character.

Also, in many cases, from a given position p in SA-space we need to jump forward to start the LF text retrieval from several bases ahead. To do so, we need to decode that position into Text-space, increment the position p, and encode back to SA-space. As we can see, full-text built-in operations given by the FM-Index are very space efficient but quite convoluted for simple text queries.

For these reasons, it is more convenient to store the full reference text together with the FM-Index. Despite the index space increasing notably, DNA text is rather suited to

being compacted – using 2 or 3 bits per character – reducing the space overhead. For instance, for the human genome with a length of approximately 3G nucleotides, the plain text could use up to 1GB of extra space. In this way, once we decode a position p into Text-space, accessing its corresponding text region becomes trivial and more efficient as all the region is stored contiguously in memory.

1.3. Sequence alignment

Sequence alignment denotes a series of bioinformatic algorithms whose purpose is to establish homology between genomic sequences. In order to do so, sequences are aligned –lined up with each other– so that the degree of similarity is maximised according to a given string distance function or score. In the case of local alignment, we search for suitably matching parts of the different sequences. In the case of global alignment, we search for the degree of similarity of the full sequences.

Sequence alignment is a general term and has multiple embodiments (as in pair-wise sequence comparisons, multiple sequence alignment, construction of evolutionary trees, etc).

Considering the problem of mapping DNA sequences against a reference genome (a.k.a mapping to a genome), mapping to a genome aims to retrieve all positions from a given reference (collection of chromosomes, contigs or so) where a given DNA sequence (relatively small and several orders of magnitude smaller than the reference) aligns (using a given distance function and error tolerance). Ultimately, the goal is to retrieve the actual genomic location which has generated the short sequenced read at hand. This represents the most common case of use and we traditionally refer to it as to searching for the best (or ”true”) match.

It is important to highlight that in general there will be more than one alignment, i.e. more than one location that might have plausibly originated the read (case of multimapping read). So, after finding all matching locations as stated before, the best candidates, that is, those showing highest sequence similarity to the original read, must be selected. In cases where no plausible locus of origin is found, or multiple equally likely candidates are found, the read must be flagged as ambiguous and possibly discarded. If one aims to retrieve all the equally distant matches having a minimum distance we say that one is performing an all-best search. If one simply wants to retrieve all matches within some error threshold we refer to an all-matches search.

Sequence alignment plays a fundamental role in many experiments like resequencing as it is responsible for mapping each read in its sequenced locus. In this way, mapping tools must pick out the most likely source location in the reference genome allowing certain error divergence from it. It is important to highlight that for common resequencing experiments with 30x coverage of the human genome, HTS Illumina machines will generate 100 nucleotide x 900 million reads. Mapped against 3,000 million bases of the human genome, this constitutes a challenging problem of performance for modern mapping tools.

1.3.1. Error model

All in all, sequence alignment is fundamentally about assessing homology between sequences. It is trivial to determine the homology of two sequences when both are the same (exact match). However, when it comes to similarity allowing errors, it is necessary to understand the very nature of the biological events that can transform a given sequence into another.

Broadly speaking, an error model is defined by what is considered to be an error (error event) and a function to score them (distance function). Given two strings (w and u) an alignment is a combination of error events that can transform w into u (or vice versa). From all the possible alignments of w into u, those with the least distance are called optimal alignments (i.e. whose error events score least using the given distance function).

In a way, optimal alignments aim to explain how to transform sequences with the least possible number of errors. Note that an error model induces an optimization problem to transform w to u – and vice versa – using a combination of error events that minimizes the distance function (alignment function).

Error events are the most basic modifications that can transform a sequence into another sequence. They represent the most common biological events that can naturally explain sequence transformations. In this way, they model chemical changes that can change one nucleotide into other e.g. sequencing errors, duplication error, variations, etc.

Most common error events are substitutions (a.k.a. mismatches; one sequence of nucleotides turned into another) and single nucleotide *indels* (insertions and deletions of single nucleotides). However, in certain situations other variations are proposed. For example, the swapping of two adjacent nucleotides (translocations).

For a given collection of error events, a distance function scores them to measure relative distance between sequences and measure homology (e.g. conservation of biological sequences) or divergency (e.g. degree of error). There is a variety of distance functions to choose from depending on the specific context of application.

Mismatch distance

allows only for substitutions and scores each one using a penalty 1 – in the simplified definition – irrespectively of the nucleotide substituted. The resulting distance is equal to the total number of nucleotides mismatching in each sequence. This distance function is limited to sequences of the same length.

Episodic distance

allows only for insertions with a penalty of 1. Unlike the rest of the distances presented here, this distance is not symmetric (i.e it may be possible to convert one sequence into another but not vice versa).

Longest common subsequence distance or Indel distance

allows only for insertions and deletions (but not for substitutions). This distance function scores each single nucleotide indel with a unitary penalty. Thus, the indel distance between two sequences is equal to the total number of nucleotides that must be deleted and inserted to transform one sequence into another.

Edit distance

In general, this distance allows for any single edit operation (i.e. substitution, insertion, and deletion) penalizing unitarily each one. So that the resulting distance is equal to the total number of edit operations that it takes to transform one sequence into another. This is one of the most commonly used distance functions due to its simplicity, versatility, and because there are many efficient alignment algorithms based on it. *Damerau–Levenshtein distance* is a variation in which transpositions (swaps of two adjacent characters) are also allowed with a penalty 1.

General scoring matrixes.

In general, it is possible to define a function to assign a penalty to each nucleotide substitution, insertion of any given length, and deletion of any length. These tailored scoring matrixes aim to statistically model the frequency of certain errors. For example, in DNA substitution mutations are of two types: transitions (i.e. inter- changes of purines (A ↔ G) or pyrimidines (C ↔ T)) and transversions (i.e. interchanges of purine for pyrimidine bases). It is known that transition mutations are much more frequent in nature. To model this, transversions would be penalized more than transitions in the scoring matrix. As a result, optimal alignments following this scoring scheme would favour transitions, leading to a more plausible transformation between sequences. Another good example is the so-called Smith-Waterman-Gotoh distance where gaps are scored according to their length.

A useful insight regarding error models is to understand that the error does not necessarily have to be placed in one of the sequences. In a more general understanding of the problem, error events are just transformations between two sequences. Hence the term *transformation operation* seems more suitable. In principle, none of the sequence has to be erroneous and alignment between both simply leads to an understanding of its structural similarities.

1.3.2. Alignment classification

Depending on the distance function, and how it scores error events with respect to their relative position, the shape of the optimal alignment may be different. There is a widely accepted classification of alignment algorithms depending on how they score deletions at the ends of the sequences – and therefore the shape of the alignment induced that is shown in figure 1-3.

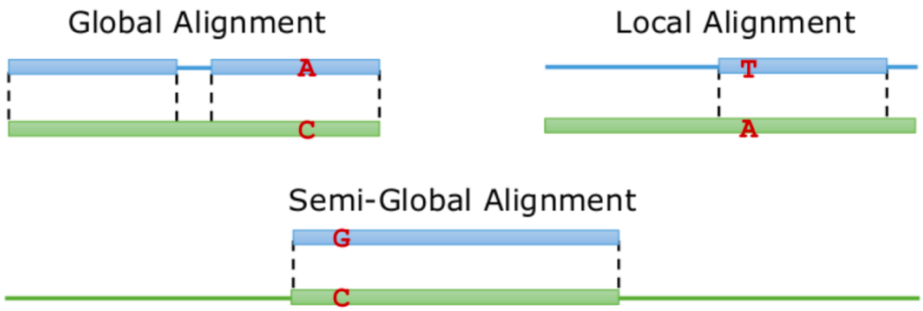

Figure 1-3. Types of sequence alignment

Global alignment

Most common alignment algorithms target global alignments which are considered the most natural form of pair-wise alignment. In this case, deletions/insertions at the ends of both sequences score as in any other position in the sequence. We say that the ends are not free and consequently must be aligned. For this reason, they are also known as "end-to-end" alignments. As a result, the final shape of the alignment spans throughout both sequences trying to match them whole.

Local alignment

By contrast, in local alignments we say that ends are free. This means that deletions at the beginning or end of the sequence do not count towards the distance function and thus trimming the sequence ends receives no penalty. These alignments tend to favor highly homologous alignments between local parts of the sequence as opposed to distant – and noisy – end-to-end alignments of the sequence.

Semi-global alignment

In cases where one of the sequences is much larger than the other, it makes sense to allow the smaller sequence to align end-to-end to a local region of the large sequence. This is a very common case when aligning relatively short HTS sequences to a relatively large reference genome. In this case, ends are free but only on the large sequence. Hence, trims on the large sequence are not considered (but they are on the smaller).

1.4. Approximate String Matching

Approximate string matching (ASM) refers to finding strings that match a pattern approximately (i.e. allowing for errors).

The most used short-read technologies produce sequences in the range 75-300 nucleotides, while a long read can be up to several thousand nucleotides long. In addition, the DNA alphabet typically has only 5 letters (that is the four bases A, C, T, G plus an additional symbol, typically N, to model uncalled/unknown bases). Furthermore, in short-read mapping one typically considers errors smaller than the 5% (but some long-read technologies can get to 30%). Yet the most striking difference is probably the size of the reference text used. In bioinformatics, genome references can be several Gbases long (for instance the human genome is approximately 3 Gbases). On the other hand,

classical problems in ASM usually deal with databases of several megabytes – an order of magnitude smaller like word databases.

1.4.1. Dynamic programming pair-wise alignment

To illustrate one of the most fundamental algorithms for pair-wise alignment, we present the solution based on dynamic programming. This approach has been rediscovered over the years within the context of several different areas. Many studies of this method have produced many fundamental theoretical bounds for the problem and suggested several approaches to avoid unnecessary computations.

Despite not being the fastest approach, it is without a shadow of a doubt the most flexible solution and the easiest to adapt to the many different distance metrics.

Following Bellman's principle of optimality, the minimum edit distance between two strings – v and w – can be computed using a dynamic programming algorithm. To avoid repeated computations given by the recursive formula, recursive calls are organised in a dynamic programing table of size (|w| + 1) × (|v| + 1). Algorithm shows how to compute this table. For instance, in following example we show the computation of the full dynamic programing table aligning the pattern: "GAGATA" against the text "GATTACA".

input : v, w strings
output: edit distance between v and w

```
1  begin
2      for i = 0 to |v| do
3          δe[i, 0] ← 0
4      end
5      for j = 1 to |w| do
6          δe[0, j] ← 0
7      end
8      for i = 1 to |v| do
9          for j = 1 to |w| do
10             δe[i, j] ← min { δe[i − 1, j] + 1
                              { δe[i, j − 1] + 1
                              { δe[i − 1, j − 1] + (vi ≠ wj)?1 : 0;
11         end
12     end
13     return δe[|v|, |w|]
14 end
```

Figure 1-4. Edit distance computation using dynamic programming

$$\delta_e(\text{"}GAGATA\text{"},\text{"}GATTACA\text{"}) = \begin{bmatrix} & - & G & A & T & T & A & C & A \\ - & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ G & 1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ A & 2 & 1 & 0 & 1 & 2 & 3 & 4 & 5 \\ G & 3 & 2 & 1 & 1 & 2 & 3 & 4 & 5 \\ A & 4 & 3 & 2 & 2 & 2 & 2 & 3 & 4 \\ T & 5 & 4 & 3 & 2 & 2 & 3 & 3 & 4 \\ A & 6 & 5 & 4 & 3 & 3 & 2 & 3 & 3 \end{bmatrix}$$

Figure 1-5: initial layout of edit distance between the example sequences

The dynamic programing algorithm fills each cell based on the content of the upper, left and upper-left neighbors. In this way, it can progress column-wise from the left most column (initial conditions) to the last column. Likewise, it can perform the computations row-wise from the upper most row to the last row. Either way, this approach runs in O(|v|||w).

However, if only the distance is required, computations can be confined to just one column – or row – and progress forward until the last column – or row – is computed and the alignment distance is known. As we can see in the example, the minimum edit distance between the string is 3 edit operations. Also, the path of the computations that lead to the minimum is highlighted in blue.

$$\Delta_e(\text{"}GAGATA\text{"},\text{"}GATTACA\text{"}) = \begin{bmatrix} & - & G & A & T & T & A & C & A \\ - & \bullet & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ G & 1 & \nwarrow & \leftarrow & 2 & 3 & 4 & 5 & 6 \\ A & 2 & 1 & \nwarrow & \nwarrow\!\!\leftarrow & 2 & 3 & 4 & 5 \\ G & 3 & 2 & 1 & \nwarrow & \nwarrow\!\!\leftarrow & 3 & 4 & 5 \\ A & 4 & 3 & 2 & 2 & 2 & \nwarrow & 3 & 4 \\ T & 5 & 4 & 3 & 2 & 2 & 3 & \nwarrow & 4 \\ A & 6 & 5 & 4 & 3 & 3 & 2 & 3 & \nwarrow \end{bmatrix}$$

Figure 1-6: path of the computation that leads to the minimum score alignment

In the previous example shown in figure 1.6, the path of the computations that lead to the minimum is highlighted in blue. In order to recover the sequence of operations that aligns one string into the other, we simply have to follow the path that updates each cell from $\delta_e(6,7)$ to $\delta_e(0,0)$ (in this context usually called edit transcript). This process is known as traceback and, as we can see, can lead to multiple possible paths – all of them equally valid.

1.4.2. Filtering algorithms

Since 1990 filtering algorithms have been studied as a very effective approximate matching technique. They base their success on the observation that, for a reasonable choice of tolerated error degree, it is easier to look for regions of the text containing

pieces of the pattern without errors rather than trying to align the pattern against every position of the text. Since looking for exact chunks of the pattern can be much faster than trying to do approximate matching with the whole pattern, filtering algorithms can quickly discard large regions of the text and perform much faster than other approaches in actual practice.

For instance, while searching for the pattern P=GATTACA up to one error, we could inspect the text and quickly discard the regions that do not contain either one of the substrings s_0 ="GATT" or s_1 ="ACA". Indeed, no matter where in the pattern the possible error is located, either s_0 or s_1 must occur in all patterns containing up to one error.
In general, filtering techniques aim to filter out as many regions of the text as possible to reduce the search space being explored down to a few candidate regions. This approach can potentially avoid inspecting the whole text, often leading to algorithms that are sublinear in the length of T in most practical cases.

A significant amount of research has been conducted in this area and a number of algorithms have been proposed for different applications and typical parameter ranges [Burkhardt 03; Weese 09; Kehr 11; Fonseca 12]. Furthermore, many filtering algorithms have been proposed and adapted for both online and indexed searches. As of today, all practical sequence alignment tools rely on filtering one way or another.

Generally speaking, all filtering algorithms are based on partitioning the pattern in order to reduce the approximate string-matching problem needed to search for the individual parts. This partition of the pattern –known as filtering scheme– induces a set of conditions that any substring of the text has to satisfy in order to be a valid match. Afterwards, using an index of the text – indexed search – or the text itself – online search – all the candidate substrings are gathered and aligned against the pattern to discover the true matches.

Traditionally, filtering algorithms are presented as just a pre-selection step to filter out regions of the text (candidate generation). In this way, filtering algorithms are unable to discover matches by themselves. Instead, a verification algorithm has to be coupled with them so as to compute the actual matches (i.e. align with distance below a given threshold). Figure 1.7 depicts the general filtering paradigm.

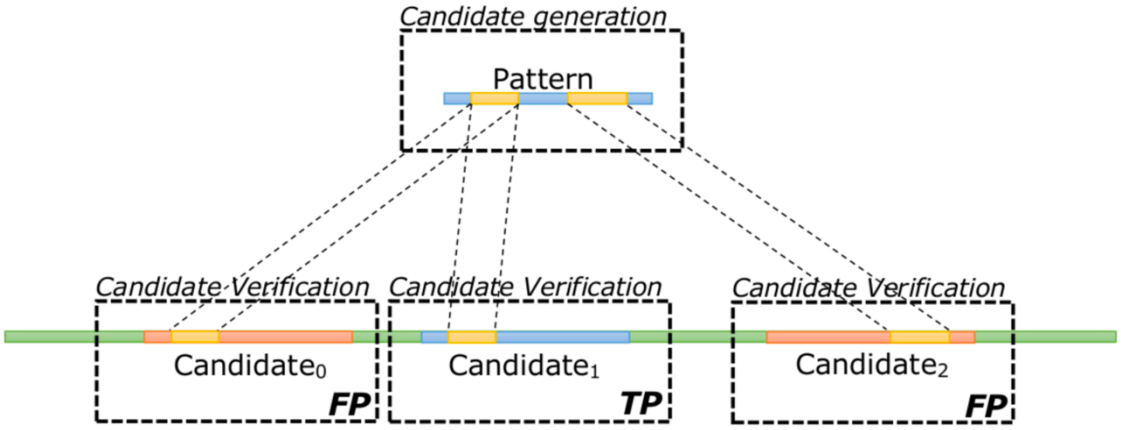

Figure 1-7: Filtering algorithm basic principles

Filtering algorithm performance strongly depends on the balance between these two stages. Very restrictive filters can produce very few candidates at the expense of intensive computations meanwhile fast and nonspecific filters can produce too many candidates. Quite often, the selection of the algorithm in the verification step is overlooked under the assumption that previous candidate generation will have led to few candidates. Nevertheless, it is extremely important to understand that both steps have different complexities. For instance, in most practical scenarios, candidate verification grows quadratically with the length of the pattern, whereas candidate generation operates linearly.

Filtering efficiency

Back to filtering algorithms, specifically to the candidate generation step, we can encounter several cases depending on the nature of the candidates reported by the filter, as in figure 1.8.

- True positive (TP); when the filter reports a candidate that aligns with the pattern.
- False positive (FP) (type I error) or "false alarm"; when the filter reports a candidate that doesn't align with the pattern.
- True negative (TN); when the filter discards a region of the text that truly doesn't align against the pattern.
- False negative (FN) (type II error); when the filter discards a region that aligns against the pattern.

		Predicted condition (Mapper)	
		Reported Match	Unmapped
True condition (Simulator)	True-Match	TP	FN
	Otherwise	FP	TN=0

Table 1.8: filtering candidates confusion table

We define the sensitivity and specificity of a filter accordingly to this.
Sensitivity (SEN) – a.k.a. true positive rate (TPR) – measures the proportion of candidates reported that are valid matches (w.r.t. the number of matches missed).

Specificity (SPC) – also called the true negative rate – measures the proportion of candidates that are correctly filter out (w.r.t. the number of valid matches incorrectly discarded).

$$SEN = \frac{TP}{TP + FN}$$

$$SPC = \frac{TN}{TN + FP}$$

Depending on the filtering algorithm, the number of discarded regions can be different – and consequently the overall computational cost. To evaluate the performance of a filter – and compare against others –, we commonly use the filtering efficiency (i.e. precision or positive predictive value; PPV) as the ratio between the number of matches found (i.e true positives) and the number of candidates produced by the algorithm (i.e. true positives + false positives).

$$f_e = \frac{TP}{TP \ + \ FP}$$

It is important to note that the maximum number of true positives reachable is independent of the algorithm – but related to the content of the text and the pattern. Therefore, it is the number of FP that we would like to minimise – without increasing the filter complexity –, understood as the capability of the filter to reduce the noise (FP).

Filtering algorithms final goal is to reduce the initial search space, like a reference genome, to a few matching regions. From this point of view, sequence alignment algorithms can be understood as filters. Each of them is given a candidate and a pattern and must discriminate their alignment

Exact filters are algorithms capable of computing whether a given string candidate w aligns a given pattern P (within a maximum distant threshold *e* and for a given distance metric *d()*). These filters are full, sensitive and specific; never fail to report a valid match (FP = 0) and never report an invalid match as so (FN = 0).

Approximate filters are not fully specific and often report false positives in the form of candidates that do not match the pattern (i.e. generate noise). There are many approximate filtering algorithms to choose from with different trade-offs between filtering efficiency and computational cost. But in general, these filters tend to be very lightweight and perform very well in practice.

In turn, we can classify approximate filters as lossless or lossy filters; depending on whether they retrieve all valid matches (i.e. true positives) or fail to report complete results

1.4.3. Exact filters. Candidate verification techniques

Most classical pairwise alignment algorithms fall into this category. Extensive research has been done in this area for many years [Needleman 70; Sankoff, 72; Sellers, 74; Wagner 74; Navarro 01]. In this way, dynamic programming (DP) approaches [Kukich 92; Ukkonen 85] explore the whole text and output only the matching positions (true positives). These types of algorithm should be classified as exact-filters as they return all matches with zero false positive rates. Despite being the most flexible solutions and delivering the best filtering efficiency, most of them are quite computationally expensive. This is the main reason to explore other kinds of filter more computationally efficient, though perhaps less accurate as they may report false positives.

Since the formulation of the first algorithms to compute the pair-wise edit distance using DP, some alternative formulations of the problem and many improved solutions have been proposed.

One of the most notorious revisions is the approach given by [Ukkonen 85] where the problem is formulated as finding the shortest path problem on a graph built on the text and the pattern. In the paper, Ukkonen presented basic theoretical properties of the DP table still exploited in modern algorithms. He formally proves that diagonals of the DP table are monotonically increasing from the upper-left to the lower-right cells. Moreover, adjacent cells value can differ at most by one. As a result, many algorithms exploit these properties towards computing only the band of the DP table where alignments up to e edit operations are confined. As a result, they drastically reduce the amount of computations needed to calculate the edit distance between two strings.

Furthermore, these properties are exploited so as to compute edit distance e in $O(e^2)$ within algorithms denoted as diagonal transition algorithms. These algorithms aim to compute in constant time the cells where the diagonal values are incremented. These diagonals – so-called "strokes" – model segments that match exactly between two strings. Afterwards, strokes are joined to compute the optimal path which leads to the optimal alignment. In [Landau 89] Landau and Vishkin propose one of the first algorithms of this kind.

Another remarkable algorithm exploiting these properties is proposed by Myers in [Myers 86]. Myers' O(nd) algorithm – unlike many others – is able to report the optimal alignment in O(|P|e) time. Many other algorithms have been proposed in this line of research.

Another formulation of the problem is given in the form of searching with an automaton [Navarro 01]. Using a non-deterministic automaton (NFA) transformation operations are modelled as transitions over a finite number of states. Each state is associated with a number of errors and a position of the pattern aligned. By introducing the letters of given string into the NFA, the final state becomes active if the string aligns the pattern. This computation can be done by means of simulating the NFA or by transforming the NFA into the corresponding deterministic automaton (DFA).

This idea was proposed in early publications like [Ukkonen 85] using a deterministic automaton. Later on, the computation of the automaton was further described and improved [Baeza-Yates 99; Wu 92]. Depending on the distance metric employed, there are solutions – like the levenshtein automaton [Schulz 02] – that perform fast in real scenarios. However, the challenging aspect of this approach is the exponential number of possible states that arises as the error increases. Hence, these algorithms are only practical for reasonably low error rates.

1.4.4. Approximate filters

In brief, these techniques are based on counting substrings in both the pattern and the candidate to assess similarity under certain error tolerance. In this way, these techniques are able to derive fixed sets of conditions that the candidates must satisfy in order to match the pattern.

Counting filters work in two basic steps. First, they build a profile based on counting occurrences of q-grams in the pattern. Afterwards, they perform the same counting procedure on the candidate text. If the counting profile is similar – depending on the maximum distance allowed – then we can conclude with some confidence that the pattern and the candidate match. These techniques stand out because they are simple, yet very fast and achieve good filtration rates.

Counting characters

In the context of online filtering, the most basic version of this family is based on counting characters [Jokinen 96]. This simple filter holds the very essence of all the filters in this family. Basically, we first count the characters in the pattern and we build a histogram of the number of times that each letter of the alphabet appears in it. Intuitively, any region of the text matching the pattern has to contain a similar count of characters. Therefore, for each region of the text we count characters. In general, any text matching the pattern with e errors must have at least $|P|-e$ characters from the pattern. Note that if the pattern and the text match perfectly, the counts must be exactly the same. Hence, we can state that if the count differs in at most e characters the candidate can be a valid match, otherwise it is not possible and can be filtered out. Note that the filter does not consider the relative ordering of the characters. Therefore, it is easy to see that it can potentially generate false positives.

The computation of the pattern profile is O(|P|) and assumed to be amortized as we check multiple candidates for each single pattern. In that way, the filter complexity is dominated by the cost of checking the candidates which operates in linear time w.r.t. the candidate length (O(|w|)).

Despite its simplicity, this simple filter can yield high filtration efficiency rates depending on the pattern length and the alphabet size. Note that a 26 character alphabet is much more diverse and restrictive than the DNA alphabet (i.e. Σ = {ACGTN}). In practice, this

filter is rarely used and it mainly serves as an introduction to logical extensions of this idea.

Counting q-grams

At this point, it seems natural to extend the idea of counting characters to count q-grams. In fact, profiling larger components of the pattern increases the specificity of the filter. Q-gram filters are based on counting q-grams to discard candidates than contain less than a given number or q-gram occurrences [Jokinen 91; Ukkonen 92]. The basic idea behind q-gram filters is depicted in figure 1-9. As the figure shows, the maximum number of q-grams that each potential error can modify depends on the q-gram length. Therefore, for a given maximum error *e* there is a minimum number of q-grams shared by the pattern and any potential match.

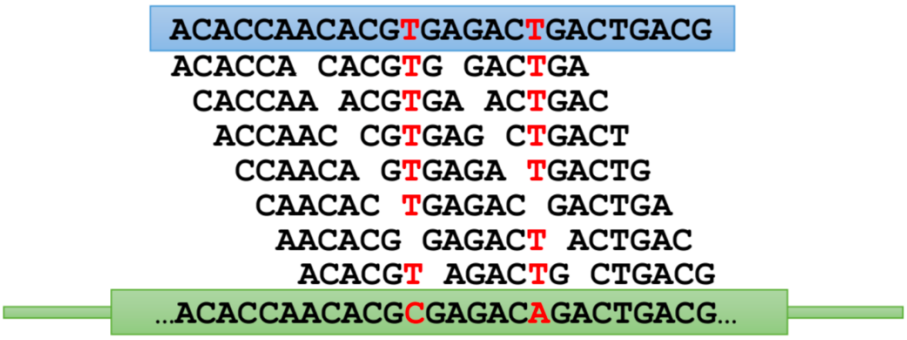

Figure 1-9: q-gram counting in a candidate

In their first formulation, q-gram filters were used in the context of online searches. The linear cost O(|w|) of counting the q-grams of the pattern (i.e. histogram or profile creating) is amortised by the cost of subsequent computations of the q-gram lemma on the candidates. For that, the algorithm applies the q-gram lemma on a sliding window that traverses every candidate and determines if the candidate is accepted or filtered out. Counters are reused between shifts of the sliding window and therefore the algorithm takes O(|w|) in the worst case.

Filtration efficiency of Q-gram filters strongly depends of the distribution of the candidates (i.e. string content), the length of the pattern, the length of the q-gram, and the maximum error. As with many other filtering techniques, there is little benefit in benchmarking its efficiency for generic random texts. Instead, it is more useful to base experiments on real application data. In general, with longer pattern lengths and longer q-gram lengths, the filter achieves better filtration efficiency. However, after reaching a certain pattern length, the filtration efficiency drops as q-grams at that length are not specific enough. Also, as the error rate increases, the filtration efficiency of q-gram filters decreases as it has to account for more errors.

It is important to note that these filters are greatly limited by the g-gram length as the memory footprint increases exponentially requiring $4^{|q|}$ counters (e.g. Using 32-bit counters, for q = 7 the counters occupy 64KB). Also note that memory consumption is

not the main limitation, but the penalties derived from randomly accessing potentially large regions of memory (e.g. cache misses, page faults, etc).

Q-gram counting technique was adapted to be used in conjunction with indexes. Many q-gram specialized indexes have been proposed to accelerate computing the q-gram profile of certain regions of the reference text (e.g. overlapping blocks of text [Burkhardt 03]). In any case, the basics of the approach remain the same; spotting a region of the text or candidate whose q-gram profile matches the one of the pattern as the g-gram lemma states. However, the indexed formulation of the algorithm allows skipping entire regions of the reference text at the expense of accesses to the index – and memory space devoted to the index. Note that in these cases the cost of building the index is also considered amortised.

Q-gram filters are often used in real tools for sequence alignment. Notable examples are SWIFT [Rasmussen 06], STELLAR [Kehr 11], RazerS [Weese 09], and RazerS 3 [Weese 12]. In fact, many practical tools incorporate in different ways the basics of q-gram counting; namely Mr. Fast [Alkan 09], Mrs. Fast [Hach 10], Hobbes [Ahmadi 12], SNAP [Zaharia 11], and many others.

2. Architectural design considerations

2.1. FM-index architectural design

One of the most important aspects of an efficient FM-index is the actual index lay-out in memory. As accessing the index is the key building block of any indexed search algorithm, the cost per access is going to be the major determinant of searching performance.

An efficient FM-index design must provide a convenient trade-off between fast-memory access and computation. In more detail, to perform an LF computation we need to access the index to fetch the text, partial counters, and C[] array and compute the occurrences for a given character. In this way, an efficient design should enable fast memory access and a lightweight rank computation.

From the memory standpoint, it is desirable to reduce the number of accesses so as to reduce the total memory bandwidth required per LF operation. But also, to efficiently access regions of memory in the order of gigabytes, it is very important to be aware of the penalties imposed by the memory wall [Wulf 95].

The computational objectives are to maximize memory locality and minimize the number of failures in all the levels of the memory hierarchy (being TLB misses the least desirable).

For that reason, architectural efforts propose ways of concentrating all the index-data associated with an LF operation in the same memory region and keep the overall index size as small as possible.

From the computation standpoint, the main bottleneck of the FM-index search is counting occurrences of a character in a given FM-Index block of BWT text. For that, several factors must be evaluated like the encoding of the characters in the text bitmaps or the number of text bitmaps between partial counters. Furthermore, depending on the encoding, some layouts allow using SIMD instructions to enhance the rank computation.

2.1.1. FM-index memory access pattern

The backward search is probably one of least memory friendly algorithms and, in practice, is bound to generate a lot of cache misses and page faults.

Experimental results show that the memory accesses of backward searches are far from depicting locality. However, these accesses are not random at all. At each step, the backward search delimits those suffixes from SA_T prefixed by the pattern searched so far Pi..m−1. Each time a character P [i − 1] is added to the search, the two sentinels delimiting the search suffixes jump to another region of the SA_T, this time, prefixed by

the new character and the previous prefix Pi−1..m−1. It is easy to see, that the accesses to the FM-index change from one region of the index to another depending on the next character added to the backward search. As presumably most of the characters of the pattern are different, each step of the backward search potentially jumps to a different, remote position of the FM-Index. Hence, the memory access pattern of the FM-Index is clearly not local at all. Furthermore, for genome-scale FM-Indexes, most accesses are bound to hit different cache blocks and a different operating system page.

The first design decision to make is the definition of an FM-Index design such that each block b_i fits entirely into a cache line. Fitting and aligning whole FM-Index blocks into a cache line objectives are:

- one cache block has to be fetched to compute the LF operation (potentially incurring in just one cache miss),
- minimize the memory bandwidth needed per LF call

Here, the key insight is that, depending on the FM-Index design, it is preferable to add padding and waste index space for the sake of aligned accesses to cache blocks and page boundaries, rather than compacting the FM-Index blocks at the expense of failing twice when accessing blocks in the middle of two cache blocks – or even two system pages.

At the same time, an overlooked feature of efficient FM-Index implementations involves the selection of architecturally friendly dimensions. For instance, note that any block size *b* for the FM-Index is theoretically possible. However, selecting *b* in such a way that T_{bwt}^i fits a computer word enables easier access and faster counting. For instance, using a 2-bits alphabet and allocating b = 32 characters in a 64 bits word.

Furthermore, the power of two block lengths leads to power of two divisions to locate the FM-Index block at each LF operation. Since the power of two divisions and modules can be implementing shifts and masks, selecting power of two block lengths can avoid frequent and expensive machine instructions as divisions. Likewise, block counters c_i can be squeezed into the domain of positions of the reference text length

2.1.2. FM-index computational needs

In terms of computation, the most expensive part of each LF call is counting the number of occurrences of a given character in a text block T^i_{bwt}. This strongly depends on the bitmap encoding and arrangement of the characters inside the block. For instance, a simple encoding of the characters packed would arrange one after another (i.e. $T^i_{bwt} = c_0 \cdot c_1 , ..., c_{b-1}$).

This naive approach not only forces several convoluted masking operations before the actual occurrences can be properly counted, but can also potentially waste bits from a computer word if the alphabet size is not a power of two (e.g. an alphabet using 3 bits would waste 2 bits per 32 bits machine word; 76MB for the human genome).

To avoid this, we could use a *character bitmap representation* – one bitmap per character – each using a computer word. In this case, counting occurrences are simplified to just shifting to remove the characters beyond the rank position and

counting ones (i.e. *popcount*). However, this representation grows in space linearly with the number of characters in the alphabet.

For that reason, a more efficient design stores the bits of each character separated in different layers according to their significance; bit-significance layers encoding. For example, in the case of 3 bits per character, this encoding would store 3 separated words containing all the character's bits grouped from least to most significant as seen in fig 2.1.

Character	Encoding
A	000
C	001
G	010
T	011
N	100
|	101

	N	G	A	T	T	A	C	A	|	T	G	T	A	A	T	C	N
$layer_0$	0	0	0	1	1	0	1	0	1	1	0	1	0	0	1	1	0
$layer_1$	0	1	0	1	1	0	0	0	0	1	1	1	0	0	1	0	0
$layer_2$	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1

Figure 2.1: Bit-significance layers encoding

In this way, counting characters becomes relatively simple by (1) negating some layers – depending on the LFc character –, (2) combining the result using bitwise AND, (3) shifting to remove the characters beyond the rank position, and (4) counting ones left in the bitmap.

2.1.3. FM-index operators

Just as important as the specific index layout, the actual implementation of the FM-Index operators plays a fundamental role in the overall performance of indexed approximated search algorithms. For this reason, careful considerations at the time of implementing these functions can have great performance impact. Additionally, basic search algorithms often present opportunities to implement tailored operators that can exploit particular conditions and perform better than single unspecific operators. For this reason, this section presents practical considerations for implementing FM-Index operations and efficient tailored operators for particular search scenarios to reduce computations and obtain better performance.

LF_c operator

Presumably, the LF_c operator is the most recurrent function within any approximated string matching algorithm based on top of the FM-Index. Within a real mapping tool, the LF_c operator can get to be called billions of times. For that reason, an efficient implementation of this function is crucial to achieve a good overall performance. Moreover, modest implementation improvements can be reflected in the overall performance. The following optimizations can easily be implemented within any FM-Index lay-out design and quantitatively help to reduce the overall number of operations to compute the LF_c

Incorporated C[c] counters

Recalling the basic LF_c definition ($LF_c(i, c) = C[c] + rank(c, i)$), to the computation of rank(c,i), we need to add the content of the accumulated occurrence array (i.e. C[c]). Note that this addition is repeated each time we call the LF_c function. For that reason, we can add the content of C[c] to each mayor counter of the FM-Index (Ci[c]) and avoid this addition whenever LF_c is called.

XOR character table

In order to avoid conditional code in the LF_c function, we can replace the conditional negation of the text bitmaps using small translation tables indexed by the character of the query. The resulting code is compact and avoids the use of conditional expressions which can lead to possible stalls in the pipeline.

Bitmap mask table

Note that the last computation of the LF_c function involves shifting unwanted bits to compute the *popcount* of the remaining ones. In order to reduce computations, we can replace the subtraction and the shift using a mask table with the precomputed masks. The resulting code reduces arithmetic operations in favor of an indexed access to a small array that can be cached easily.

Power-of-two divisions/modulus

Divisions and modulus operations by powers of two can easily be replaced for shifts and masking instructions. Indeed, a modern compiler is expected to perform this substitution on-the-fly. As a result, some of the most computationally intensive instructions are avoided and much faster code can be generated.

Specialised popcount hardware instruction

Out of all the instructions executed within the LF_c function, the *popcount* operation can be the most time consuming. In fact, this operation is so computationally intensive and recurrent in computer applications that many hardware architectures implement tailored instructions for it. In the case of Intel architectures, the SSE 4.2 instruction set extensions offer *popcount* hardware instruction that can greatly accelerate this computation. LF_c implementations using it can experiment a speedup of almost a 30% *popcount* [Warren 13].

2.2. Sequence filtering accelerating computations

2.2.1. Block-wise computations

One of the most widely known approaches to improving the computation of the DP table is the so-called Four-Russians technique [Kronrod 70]. Their approach pre-computes all possible combinations of small DP tables of size *b* (i.e. look-up table of all possible DP blocks of size *b*). Afterwards the algorithms proceed block-wise computing the full matrix taking advantage of precomputed blocks; thus gaining a factor of *b*. Since neighboring cells differ at most by one, the DP is encoded using horizontal– or vertical – differences ($\delta_e(i,j) \in \{-1,0,+1\}$). This makes a total of $(3|\Sigma|)^{2b}$possible input combinations to each precomputed block. Choosing b=O(log(|P|)) the complexity gets reduced to $O(\frac{|P|*|T|}{log(|P|)})$. In short, assuming equal text and pattern length n, the complexity drops from $O(n^2)$ to $O(n^2/\log(n))$

2.2.2. Bit-parallel algorithms

Another very successful technique employed to accelerate the computation of pair-wise alignment is to exploit the intrinsic parallelism of the computer instructions. Nowadays, any computer architecture offers simple arithmetical and logical instructions that can be understood as parallel operators over a set of bits. These operations can be performed using 32 or 64 bit words and, depending on the architecture, using SIMD instruction from 128 to 512 bits long.

Here, the key insight is to encode the problem conveniently so as to benefit from simple bit-wise instructions. Algorithms based on this idea are able to compute several cells of the DP table – or states of the NFA – at once, enabling much faster computation of pair-wise alignments.

Most notably, Myers in [Myers 99] proposed the computation of the DP matrix encoding columns using bit-words and using bit-wise operations to fill the matrix. The so-called Bit-Parallel Myers (BPM) takes advantage of the intrinsic parallelism within bit-wise operations of any computer architecture (i.e logical, shift, and addition operations).

In this approach, columns of the DP matrix are encoded in differences among themselves. As explained before, this reduces the number of possible values of each cell to the values {−1, 0, +1}. Then, each cell of the matrix is modelled as a logic block. In this way, the basic step of the processing takes a column of the DP and produces the next column using bit-wise operations. This process iterates so as to compute the whole matrix chunk-wise meanwhile the cells of each column are computed intrinsically in parallel. Low-level details of the algorithm make it one of the quickest bit-wise algorithms of its type [Myers 99; Hyyro 03].

The BPM algorithm has a worst-case complexity of O(|P||w|/b) – where b is the length of the computer word used – and O(e|w|/b) on average as cut-off conditions can be easily implemented within. What is more, this algorithm performs better than others of its kind with long patterns. For this reason, it is often the choice of many real mappers [Marco-Sola 12; Siragusa 13] based on filtering to verify candidates. Additionally, in [Hyyro 02] an improved version of the BPM algorithm was proposed so as to reduce the overall bit-wise operations performed.

2.2.3. SIMD algorithms

Using general scores, some techniques have been proposed to exploit SIMD instructions in order to compute several cells of the DP matrix at once. The key idea is to allocate several cells into a computer register and compute the DP matrix by blocks [Alpern 95]. In this way, these algorithms differ in that they arrange the cells into the registers (i.e. by columns [Rognes 00], anti-diagonals [Wozniak 97] or interleaved [Rognes 11]). The main challenge of these methods is to incorporate all the dependencies in the SIMD computation without introducing mayor slowdowns. In this way, Farrar's approach in [Farrar 07] proves to be the most efficient, achieving from 2x-8x folds against previous approaches. In practice, this method is widely used and some mappers based on SWG distance implement it [Langmead 12; Li 13].

3. Hardware support for bioinformatics applications

Hardware accelerators have been proposed to accelerate genomic analysis pipelines. Many popular tools in the processing of common bioinformatics pipelines like NGS short read data processing with BWA-MEM, GATK for variant calling, SMEM for seeding, Smith-Waterman for extending and PairHMM for variant calling.

Accelerators can be implemented in customized computing technologies, such as GPU, FPGA or ASIC. [Luk 17] give a comprehensive review on FPGA accelerator for genomics presenting an FM-index implementation in FPGA. [Ahmed 16] gives a good overview comparing seed-and-extend techniques in DNA read alignment algorithms.

In addition to BWA-MEM+GATK, certain cancer genomics pipelines also feature alignment refinement before variant calling. The alignment refinement pipeline typically includes sorting, duplicate marking, INDEL realignment, and base quality score recalibration. [Wu 18] introduce an FPGA accelerated INDEL realignment accelerator in the AWS Cloud that accelerates INDEL realignment by 81x and reduces cost by 32x.

Specific hardware accelerators for genomics has been a recent field of advances. Darwin [Turakhia 18] is a genomic processing accelerator that accelerates GraphMap for third generation genomic data processing by 1000x on long read genome assembly. GenAx [Fujiki 18] is also a genomic sequencing accelerator that accelerates BWA-MEM by 31.7x with SMEM plus hash-based seeding and automata-based extending stages.

4. References

[Ahmadi 12]: A. Ahmadi, A. Behm, N. Honnalli, C. Li, L. Weng, and X. Xie. Hobbes: optimized gram-based methods for efficient read alignment. Nucleic acids research, 40(6):e41–e41, 2012.
[Ahmed 16]: Nauman Ahmed,Koen Bertels, and Zaid Al-Ars.2016.A comparison of seed-and-extend techniques in modern DNA read alignment algorithms. In Bioinformatics and Biomedicine (BIBM), 2016 IEEE International Conference on. IEEE, 1421–1428.
[Alkan 09]: C. Alkan, J. M. Kidd, T. Marques-Bonet, G. Aksay, F. Antonacci, F. Hormozdiari, J. O. Kitzman, C. Baker, M. Malig, O. Mutlu, et al. Personalized copy number and segmental duplication maps using next-generation sequencing. Nature genetics, 41(10):1061–1067, 2009.
[Apern 95]: B. Alpern, L. Carter, and K. Su Gatlin. Microparallelism and high-performance protein matching. In Proceedings of the 1995 ACM/IEEE conference on Supercomputing, page 24. ACM, 1995.
[Baeza-Yates 99]: Baeza-Yates and R. G. Navarro. Faster approximate string matching. Algorithmica, 23(2):127–158, 1999.
[Burkhardt 03]: S. Burkhardt and J. Karkkainen. Better filtering with gapped q-grams. Fundamenta informaticae, 56(1-2):51–70, 2003.
[Burrows 94]: M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.
[Farrar 07]: M. Farrar. Striped smith–waterman speeds database searches six times over other simd implemen- tations. Bioinformatics, 23(2):156–161, 2007.
[Ferragina 00]: P. Ferragina and G. Manzini. Opportunistic data structures with applications. In Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, pages 390–398. IEEE, 2000.
[Ferragina 09]: P. Ferragina, R. Gonz´alez, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. Journal of Experimental Algorithmics (JEA), 13:12, 2009.
[Fonseca 12]: N. A. Fonseca, J. Rung, A. Brazma, and J. C. Marioni. Tools for mapping high-throughput sequencing data. Bioinformatics, page bts605, 2012.
[Fujiki 18]: Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A genome sequencing ac-celerator. In Proceedings of the 45th Annual International Symposium on Computer Architecture. IEEE Press, 69–82
[Gargis 15]: A. S. Gargis, L. Kalman, D. P. Bick, C. Da Silva, D. P. Dimmock, B. H. Funke, S. Gowrisankar, M. R. Hegde, S. Kulkarni, C. E. Mason, et al. Good laboratory practice for clinical next- generation sequencing informatics pipelines. Nature biotechnology, 33(7):689–693, 2015.
[Grossi 03]: R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, pages 841–850. Society for Industrial and Applied Mathematics, 2003.
[Hach 10]: F. Hach, F. Hormozdiari, C. Alkan, F. Hormozdiari, I. Birol, E. E. Eichler, and S. C. Sahinalp. mrsfast: a cache-oblivious algorithm for short-read mapping. Nature methods, 7(8):576–577, 2010.

[Hwang 15]: S. Hwang, E. Kim, I. Lee, and E. M. Marcotte. Systematic comparison of variant calling pipelines using gold standard personal exome variants. Scientific reports, 5, 2015.
[Hyyro 02]: H. Hyyro and G. Navarro. Faster bit-parallel approximate string matching. In Annual Symposiumon Combinatorial Pattern Matching, pages 203–224. Springer, 2002.
[Hyyro 03]: H. Hyyro. A bit-vector algorithm for computing levenshtein and damerau edit distances. Nord. J. Comput., 10(1):29–39, 2003.
[Jokinen 91]: P. Jokinen and E. Ukkonen. Two algorithms for approxmate string matching in static texts. In International Symposium on Mathematical Foundations of Computer Science, pages 240–248. Springer, 1991.
[Jokinen 96]: P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. Software: Practice and Experience, 26(12):1439–1458, 1996.
[Kehr 11]: B. Kehr, D. Weese, and K. Reinert. Stellar: fast and exact local alignments. BMC bioinformatics, 12(9):S15, 2011.
[Kronrod 70]: M. Kronrod, V. Arlazarov, E. Dinic, and I. Faradzev. On economic construction of the transitive closure of a direct graph. In Sov. Math (Doklady), volume 11, pages 1209–1210, 1970.
[Kukich 92]: K. Kukich. Techniques for automatically correcting words in text. ACM Computing Surveys (CSUR), 24(4):377–439, 1992.
[Landau 89]: G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. Journal of algorithms, 10(2):157–169, 1989.
[Langmead 12]: B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. Nature methods, 9 (4):357–359, 2012.
[Li 13]: H. Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. arXiv preprint arXiv:1303.3997, 2013.
[Loman 12]: N. J. Loman, R. V. Misra, T. J. Dallman, C. Constantinidou, S. E. Gharbia, J. Wain, and M. J. Pallen. Performance comparison of benchtop high-throughput sequencing platforms. Nature biotechnology, 30(5):434–439, 2012.
[Luk 17]: Ho-Cheung Ng, Shuanglong Liu, and Wayne Luk. 2017. Reconfigurable accelera- tion of genetic sequence alignment: A survey of two decades of efforts. In Field Programmable Logic and Applications (FPL), 2017 27th International Conference on. IEEE, 1–8.
[Marco-sola 12]: S. Marco-Sola, M. Sammeth, R. Guig´o, and P. Ribeca. The gem mapper: fast, accurate and versatile alignment by filtration. Nature methods, 9(12):1185–1188, 2012
[Marx 13]: V. Marx. Biology: The big challenges of big data. Nature, 498(7453):255–260, 2013.
[Myers 86]: E. W. Myers. Ano (nd) difference algorithm and its variations. Algorithmica, 1(1-4):251–266, 1986.
[Myers 99]: G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic pro- gramming. Journal of the ACM (JACM), 46(3):395–415, 1999.
[Navarro 01]: G. Navarro. A guided tour to approximate string matching. ACM computing surveys (CSUR), 33(1):31–88, 2001.
[Navarro 07]: G. Navarro and V. M˙akinen. Compressed full-text indexes. ACM Computing Surveys (CSUR), 39 (1):2, 2007.

[Needleman 70]: S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of molecular biology, 48(3):443–453, 1970.
[Rasmussen 06]: K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ε-matches over a given length. Journal of Computational Biology, 13(2):296–308, 2006.
[Reuter 15]: J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. Molecular cell, 58(4):586–597, 2015.
[Rognes 00]: T. Rognes and E. Seeberg. Six-fold speed-up of smith–waterman sequence database searches using parallel processing on common microprocessors. Bioinformatics, 16(8):699–706, 2000.
[Rognes 11]: T. Rognes. Faster smith-waterman database searches with inter-sequence simd parallelisation.BMC bioinformatics, 12(1):221, 2011.
[Sankoff 72]:D. Sankoff. Matching sequences under deletion/insertion constraints. Proceedings of the National Academy of Sciences, 69(1):4–6, 1972.
[Schulz 02]: K. U. Schulz and S. Mihov. Fast string correction with levenshtein automata. International Journal on Document Analysis and Recognition, 5(1):67–85, 2002.
[Sellers 74]: P. H. Sellers. On the theory and computation of evolutionary distances. SIAM Journal on Applied Mathematics, 26(4):787–793, 1974.
[Siragusa 13]: E. Siragusa, D. Weese, and K. Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. Nucleic acids research, 41(7):e78–e78, 2013.
[Siren 08]: J. Siren, N. Valimaki, V. Makinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In International Symposium on String Processing and Information Retrieval, pages 164–175. Springer, 2008.
[Turakhia 18]: Yatish Turakhia, Gill Bejerano, and William J Dally. 2018. Darwin: A Genomics Co-processor Provides up to 15,000 X Acceleration on Long Read Assembly. In Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, 199–213.
[Ukkonen 85]: E. Ukkonen. Algorithms for approximate string matching. Information and control, 64(1-3):100– 118, 1985a.
[Ukkonen 92]: E. Ukkonen. Approximate string-matching with q-grams and maximal matches. Theoretical com-puter science, 92(1):191–211, 1992.
[Wagner 74]: R. A. Wagner and M. J. Fischer. The string-to-string correction problem. Journal of the ACM (JACM), 21(1):168–173, 1974.
[Warren 13]: H. S. Warren. Hacker's delight. Pearson Education, 2013.
[Weese 09]: D. Weese, A.-K. Emde, T. Rausch, A. Doring, and K. Reinert. Razers—fast read mapping with sensitivity control. Genome research, 19(9):1646–1654, 2009.
[Weese 12]: D. Weese, M. Holtgrewe, and K. Reinert. Razers 3: faster, fully sensitive read mapping. Bioinformatics, 28(20):2592–2599, 2012.
[Wozniak 97]: A. Wozniak. Using video-oriented instructions to speed up sequence comparison. Computer applications in the biosciences: CABIOS, 13(2):145–150, 1997.

[Wu 18]: Yuanrong Wang,Xueqi Li,Dawei Zang,Guangming Tan,and Ninghui Sun.2018. Accelerating FM-index Search for Genomic Data Processing. In Proceedings of the 47th International Conference on Parallel Processing. ACM, 65.
[Wu 92]: S. Wu and U. Manber. Fast text searching: allowing errors. Communications of the ACM, 35(10):83–91, 1992.
[Wulf 95]: W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. ACM SIGARCH computer architecture news, 23(1):20–24, 1995.
[Zaharia 11]: M. Zaharia, W. J. Bolosky, K. Curtis, A. Fox, D. Patterson, S. Shenker, I. Stoica, R. M. Karp, and T. Sittler. Faster and more accurate sequence alignment with snap. arXiv preprint arXiv:1111.5572, 2011.