International Conference on Computational Science, ICCS 2013

n-step FM-Index for faster pattern matching

Alejandro Chacón*, Juan Carlos Moure, Antonio Espinosa, Porfidio Hernández

Computer Architecture and Operating Systems Department, Escola d’Enginyeria, Universitat Autònoma de Barcelona Campus UAB, Bellaterra 08193, Spain

Abstract

Fast pattern matching is a requirement for many problems, specially for bioinformatics sequence analysis like short read mapping applications. This work presents a variation of the FM-index method, denoted n-step FM-index, that is applied in exact match genome search.

We propose an alternative two-dimensional FM-index structure that allows backward-search navigation giving steps of \( n \) symbols at a time. The main advantages of this arrangement are the reduction of the computational work, but most importantly, the reduction by \( n \) of the chain of dependent data accesses, and the increase in the temporal locality of the data access pattern. This benefit comes at the expense of increasing the total amount of data required for the index.

We present an in-depth performance analysis of a multi-core implementation of the algorithm using large references (up to 1.5G). We identify memory latency as the major performance limiter for single-thread execution and memory bandwidth for multi-thread execution. Our proposal provides speedups ranging from 1.4× to 2.4×, when there is no limitation on DRAM capacity.

We also analyse the trade-off of compacting the proposed data structure in order to reduce memory capacity requirements, now at the expense of increasing execution time. An extra 33% of DRAM space allows our proposal to improve performance by 1.2×, while doubling DRAM size enables an additional 1.5×.

Our proposal of \( n \)-step algorithm provides an alternative for pseudo-random memory access algorithms to be redesigned to scale in current and future computer systems.

Keywords: Pattern Matching; FM-index; Parallel Algorithms; Performance Analysis; Multicore Processors;

1. Introduction

Next-generation DNA sequencing technologies produce over 600 gigabases of DNA sequencing reads from a single instrument run. Sequence alignment is required for downstream data analysis. Resequencing projects need to align (most commonly known as mapping) a long list of short sequencing reads to a reference genome, usually with a low number of differences from this reference. For the case of de novo sequence assembly, reads must be aligned to other reads in order to recreate a genome. This large amount of data to process demands fast and efficient pattern matching algorithms. Recent sequence alignment software tools, like Bowtie[1], SOAP[2], BWA[3], and GEM[4], use a relatively new suffix tree-based algorithm based on the Burrows-Wheeler Transform (BWT)[5] that is called FM-index[6].

*Corresponding author. Tel.: +34-93-581-1990 ; fax: +34-93-581-2478.
E-mail address: {alejandro.chacon, juancarlos.moure, antonio.espinosa, porfidio.hernandez}@uab.es.

1877-0509 © 2013 The Authors. Published by Elsevier B.V.
Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.170
FM-index implements a backward search mechanism on top of BWT, which allows finding exact pattern matches in a number of steps that is linear with the length of the searched pattern, and independent from the size of the reference sequence. In addition to its good computational complexity, it achieves high compression ratios, allowing to index the full human genome into less than 1.5 GB of memory space.

Improving execution throughput on current computer systems, composed of multiple sockets of multicore CPUs and a hierarchical memory architecture, relies on finding parallelism in the application algorithm and performing data accesses with high reference locality. Temporal locality in the data access stream is exploited to reduce access latency and save both memory bandwidth and energy consumption. Spatial locality is used to amortize costly data accesses to DRAM in the form of read and write bursts. Finally, memory-level parallelism helps tolerating high memory access latencies.

Sequence alignment algorithms usually offer plenty of thread and memory-level parallelism, easily obtained by performing multiple simultaneous pattern searches (one for each short read sequence) on the same reference sequence. This parallelism allows exploiting all the execution cores in the system and can be combined with multithreading and prefetching techniques to better exploit the available memory bandwidth. The FM-index search algorithm, though, lacks both temporal and spatial data access locality. Its good properties of high compression and fast operation are at the cost of a pseudo-random data access pattern. This lack of locality creates severe difficulties to make an efficient usage of the memory hierarchy, then limiting its potential scalability.

In order to improve the search scalability, we propose a generalization of the FM-index search algorithm, called n-step FM-index. It basically divides the number of search steps by n, at the expense of increasing the total amount of data required for the index. Each search step is computationally more complex and the whole amount of data read on each search operation is barely reduced, but the spatial locality of memory accesses is improved, and this also improves performance.

Our main contributions can be summarized as follows:

- We introduce an algorithmic variation of the FM-index search that reduces the number of search steps in order to improve execution time, at the expense of increasing the size of the BWT data structure.
- We present an in-depth analysis of performance of a multi-core implementation of the algorithm using large indexes (up to 1.5 G). Speedups from 1.4× to 2.4× are reached. We identify memory latency and memory bandwidth as the major performance limiters.
- We analyse the trade-off of compacting the BWT data structure to reduce memory capacity requirements at the expense of increasing execution time. An extra 33% of DRAM space allows our proposal to improve performance by 1.2×, while doubling DRAM size enables an additional 1.5×.

In section 2 we are going to describe the FM-index structure and operation. Section 3 introduces our proposal of increasing data locality by n-step FM-index search. In section 4, we provide experimentation results for the execution of the proposal on a multicore system. Section 5 discusses related work and, finally, section 6 summarizes results obtained and future work.

2. Background: single-step Ferragina-Manzini Index (FM-Index)

Indexing a reference sequence or string is a method to accelerate pattern search. The time spent on creating the index is amortized when a large enough number of searches is presented. Total memory capacity requirements to store this index must also be considered. As mentioned above, FM-index is the preferred indexing method used in most sequence alignment software tools. Next, we introduce the fundamental concepts behind the FM-index structure and operation.

2.1. Notation and basic concepts

Let $S = S_0S_1\cdots S_{|S|-1}$ be a sequence or string over an alphabet $\Sigma$, where $S_i$ is the $i^{th}$ symbol of the string. $S_{i:j} = S_iS_{i+1}\cdots S_j$ is a substring of $S$. $S_i = S_{i:j}$ is a suffix of $S$ starting at position $i$. Representing DNA only requires the symbols {A,C,G,T}. We use $R$ to denote the reference sequence and $Q$ to denote a query sequence. We suppose that the length of $R$ ($|R|$) is much higher than the length of $Q$ ($|Q|$). The exact matching problem consists of finding all the occurrences of $Q$ into $R$, i.e., the position of the substrings of $R$ that are equal to $Q$. 
2.2. Forward- and Backward-Search

The suffix-trie of a string $R$ is a tree-like data structure storing the sorted suffixes of $R$. The path from the root of the suffix-trie to a leaf node defines a single suffix. Each leaf node represents a single suffix and each internal node determines an interval of lexicographically consecutive suffixes.

Fig.1.a shows an example of suffix-trie and illustrates the searching process. It consists of traversing the suffix-trie, starting from the root, matching successive symbols of a query $Q$ with the visited nodes. Each symbol $Q_{[i]}$ involves an index lookup and represents a single search step. Each step progressively bounds the interval of matching suffixes. The query can be forward-searched or backward-searched [7]. Forward and backward indicate the order in which the symbols of $Q$ are used, from $Q_{[0]}$ to $Q_{[|Q|-1]}$ or vice versa. A backward-search uses the suffix-trie of the reversed reference string $R^r = R_{[|R|-1]}R_{[|R|-2]} \cdots R_{[0]}$. The example of Fig.1.a performs a backward search involving $|Q|=4$ steps. The result is a sub-tree with a single leaf node, which represents a single match.

![Fig. 1. Backward-search of $Q=acaa$ in $R=acaaacatat$ using the suffix-trie of $R^r$ (reverse of $R$) a) Single-step search. b) 2-step search.]

2.3. Suffix-Array and Suffix-Array Intervals

The Suffix-Array ($SA$) of $R$ is the permutation of the sorted suffixes of $R$, where each suffix $R_i$ is represented by its starting position, $i$. To simplify the search operation, the symbol $S$, lexicographically lower than all symbols in $\Sigma$, is generally appended at the end of $R$ before generating $SA$. In our example $R=acaaacatat$, the sorted suffixes are $R_{[10]} R_{[5]} R_{[7]} R_{[3]} R_{[8]} R_{[6]} R_{[1]} R_{[5]} R_{[9]} R_{[7]}$ and $SA=[10, 2, 3, 0, 4, 8, 6, 1, 5, 9, 7]$.

An interval in $SA$ represents a sequence of lexicographically consecutive suffixes of $R$. The interval $(low, high)$ of $SA$ represents suffixes $R_{SA[low]}, R_{SA[low+1]} \cdots R_{SA[high-1]}$. An $SA$ interval can represent the result of searching $Q$ in $R$. If $(low < high)$, then the $|Q|$ starting symbols of suffixes in the $SA$ interval correspond to exact matches of $Q$ in $R$. $SA[low], SA[low+1] \cdots SA[high-1]$ identify the starting position of each matching occurrence in $R$. $(low == high)$ indicates that $Q$ does not occur in $R$. In other words, search solutions can be represented in the $SA$ domain, using $SA$ coordinates, instead of the $R$ domain. The result of the search operation in Fig.1 is the $SA$ interval $(3, 4)$.

2.4. The Burrows-Wheeler Transform (BWT)

The FM-Index is built upon the Burrows-Wheeler Transform (BWT) [5]. The BWT of string $R$, denoted $B$, is a permutation of the symbols of $R$ that allows performing the same search operations as in a suffix-trie, using an operation called LF-Mapping [6]. $B$ is a self-index, meaning that it allows searching patterns in $R$ without requiring $R$ and $SA$. This kind of indexes reduce memory storage requirements compared to other index structures [8] [9] [10]. Fig.2.a shows how to generate $B$ from $R$ and its suffix array, $SA$. Each position of $B$ is computed as $B_{[i]} = R_{[SA[i]-1] \mod |\Sigma|}$. For example, the $BWT$ of $R=acaaacatatS$ is $B=tca$satcaaaa.
Algorithm 1: 1-step Backward Search

<table>
<thead>
<tr>
<th>input</th>
<th>Q= query, F= FM-index of R, d= distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>high-low occurrences in R.</td>
</tr>
</tbody>
</table>

begin

(low, high) ← (0, |R|)

for i = |Q|−1 to 0 do

| low ← LF-Mapping(Q[i], low) |
| high ← LF-Mapping(Q[i], high) |

end

return (low, high)

end

Function LF-Mapping(s, pos)

begin

entry ← F[pos/d]

big_cnt ← entry.LF[s] 

small_cnt ← Occ(entry.BMP, s, pos mod d)

return big_cnt + small_cnt

end

2.5. Single-step FM-Index search

The FM-Index [6] was proposed as a way to traverse a BWT representation in a tree-like way. It combines the compression advantage of BWT with the low computational complexity of tree-like searching algorithms. Algorithm 1 describes the backward-search of Q in a BWT index B. The initial search interval is set to (low=0, high=|R|), the whole set of R suffixes. The following loop performs |Q| search steps. Each step applies the LF-Mapping operation to both points of the interval using a new symbol Q[i], reducing the search space until generating the final interval.

The LF-Mapping operation is classically defined as the addition of two counting functions: C() + Occ(). C(B, s) counts the number of occurrences in B of symbols that are lexicographically lower than a given symbol s. Occ(B, s, pos) counts the number of times a symbol s appears in B_{pos−1} (before position pos). An improvement proposed by [6] is to precompute or memoize C() and Occ(). As a trade-off between memory space and time, only the Occ() values for positions that are multiple of some integer distance d are stored.

Fig. 2.b shows our proposal for the FM-index data structure, denoted by F. B is divided into blocks of d consecutive symbols. The i-th block of B, denoted as Bblock[i], is the substring B_{i·d,(i+1)·d−1}, and there is an entry in F, F[i], for each Bblock[i]. F contains |B|/d entries, and each entry contains Σ counters LF and a bitmap representation of Bblock[i], denoted BMP, which encodes each symbol using log_2 Σ bits. d allows varying the compression ratio of F, and defines a trade-off between memory and computation requirements: the larger d, the lower the size of F but the higher the number of memory load and counting operations. The $ symbol is not encoded, but its position in B is annotated and considered when performing the LF-Mapping operation, as in [3].

The LF-Mapping function shown in Algorithm 1 starts calculating the entry F[i] containing the BWT character in position pos. Then, it reads the contents of F[i] to generate a new position. Our F structure memoizes into LF counters the value LF-Mapping(s, i·d), corresponding to the first position of Bblock[i], saving one memory load and one addition operation. Occ() is equal to Occ(), but works on BMP, the bit-level representation of Bblock[i], which is optimized for fast counting using the popcount instruction available in the SSE4.1 extension of the x86 ISA.
3. Proposal: \textit{n}-step FM-index

This work presents a variation of the FM-index, denoted \textit{n}-step FM-index, that exhibits higher spatial locality in its data access pattern, and is more suitable to achieve higher performance in current computers. Next, we introduce the design principles and then give more implementation details.

3.1. \textit{n}-step BWT

We propose a two-dimensional BWT of a string \( R \), denoted \( n \times B \) (also \( n \times B \)) that allows a backward-search navigation giving steps of \( n \) symbols at a time. The idea is illustrated in Fig. 1.b, with a suffix-trie that allows searches in steps of 2 symbols. \( n \times B \) is composed of \( n \) strings, \( \{B_1[1], B_2[1], \ldots , B_n[1]\} \), where each string is the BWT of \( R \) with a different depth. \( B_1[1] \) is a different notation for \( B \). The \( i^{th} \) position of \( B_j[1] \) is computed as \( B_j[1][i] = R_1[S_1[A_1-1-j] \mod |R|] \). Fig. 3.a illustrates the process of generating \( n \times B \), which contains \( n \times |R| \) symbols. For example, the \( n \times B \) transform of \( R = acaaacatat $ \) is \( \{B_1[1], B_2[1]\} = \{tca\$atcaaaa, aactaaas$\).

Fig. 3. (a) \( n \times B \ (n \times B) \) is the \( n \)-step BWT generated from \( R \) and \( SA \); (b) \( n \times F \ (n \times F) \) is the \( n \)-step FM-index of \( n \times B \).

3.2. \textit{n}-step FM-Index search

Figure 3.b shows the generation of \( n \times F \) (\( n \)-step FM-index) from \( n \times B \). Again, one entry is generated per \( n \times B \) block of size \( d \). Each entry contains \( |\Sigma|n \) \( LF \) counters and \( n \cdot \log_2|\Sigma| \) bitmaps of size \( d \). The size devoted to counters grows exponentially with \( n \), while the size devoted to bitmaps grows linearly with \( n \). For example, with \( d = 32 \), \( |R| = 1.5G \) and \( n = \{1, 2, 3, 4\} \) the sizes of \( n \times F \) are \( \{1.0GB, 3.6GB, 12.8GB, 48.3GB\} \). Anyway, as we will analyse later in this paper, increasing \( d \) allows reducing the memory requirements for counters.

Algorithm 2 shows that \( n \) consecutive symbols of \( Q \) are grouped into a string \( str \) on each loop iteration. Then, \( str \) indexes the appropriate \( LF \) counter, and \( n \_Occ \) counts the occurrences of \( str \) in the actual \( n \_F \) block. The function uses the same bit-level optimizations described in section 2.5, but requires \( n \) times more operations. Also, since \( S \) appears in \( n \) positions of \( n \times B \), \( n \) conditions must be checked per search step but, since the number of search steps is reduced by \( n \), the total number of comparisons per query will remain the same.

A final corner case happens when the last search step involves less than \( n \) symbols, say \( k \). The solution is to aggregate all the \( LF \) counters matching with the \( k \) initial symbols of \( str \). Additionally, \( n \_Occ \) must count occurrences ignoring the last \( n - k \) symbols of \( str \).

\textbf{Algorithm 2:} \textit{n}-step Backward Search (+ special case is required when \(|Q|\) is not multiple of \( n \))

| input : \( Q = \) query, \( n \_F = \) \( n \)-step FM-index, \( d = \) distance |
| output: high-low occurrences in \( R \). |

\textbf{begin} \( (low, high) \leftarrow (0, |R|) \)

\textbf{for} \( i = |Q| \) to \( n \) \textit{step} \(-n\) \textbf{do}

\( str \leftarrow Q_{n \_i - 1} \)

\( low \leftarrow n \_LF \_Mapping(str, low) \)

\( high \leftarrow n \_LF \_Mapping(str, high) \)

\textbf{end}

\textbf{return} \( (low, high) \)

\textbf{Function} \( n \_LF \_Mapping(str, pos) \)

\textbf{begin}

\( entry \leftarrow n \_F[\text{pos}/d] \)

\( big \_cnt \leftarrow entry \_LF[\text{str}] \)

\( small \_cnt \leftarrow n \_Occ[entry \_BMP, \text{str, pos} \mod d] \)

\textbf{return} \( big \_cnt + small \_cnt \)

\textbf{end}
4. Experimental Results

We have executed the proposed $n$-step FM-index algorithm in a multicore CPU system in order to evaluate its performance and identify architectural features that contribute to that performance.

4.1. Experimental Setup and Methodology

The computer system is a dual-socket Intel Xeon E5645, each socket containing 6 Westmere cores at 2.4 GHz, and each core being able to execute 2 hardware threads using hyperthreading technology. Therefore, it can simultaneously execute up to 24 threads by hardware. 6 DIMMs of 8 GiB 1333-MHz DDR3 RAM memory per 2 sockets provide 96 GiB of storage capacity. The Last Level Cache (LLC) provides 12 MiB of shared storage for all the cores in the same socket. Each socket provides a peak external memory bandwidth of 32 GB/sec, and the Quickpath interconnection (QPI) between the two sockets provides a peak bandwidth of 11.72 GB/sec per link direction (a total of 23.44 GB/sec).

The genome references used in our experiments are built from the Human genome (GRCh37). The sequencing reads used as queries are generated from pseudo-random positions on the genome reference using the application script genreads from the Mummer project [11].

The factors analyzed in our experiments are: (1) the number of steps used by the FM-index, $n = \{1, 2, 3, 4\}$; (2) the reference size, $|R| = \{500, 2K, 5K, 20K, 60K, 200K, 600K, 2M, 8M, 50M, 100M, 400M, 750M, 1500M\}$; (3) the number of threads in which the execution is divided, $\text{thr} = \{1, 12, 24\}$; and (4) the distance between counters in the FM-index representation, $d = \{32, 64, 128, 256\}$. When not mentioned, we assume $\text{thr}=24$, $d=32$ and $|R|=1.5G$, which represents the best performing configuration.

The parameters fixed in our experiments are the total number of queries, set to 10 million, and the query length, $|Q| = 120$. We have checked that, as expected, execution time grows linearly with the total number of queries. Therefore, using a larger number of queries does not provide any additional information. Also, experiments with shorter reads, from 60 to 120 bp, produce similar performance results and are skipped from our analysis.

Each execution experiment is repeated 10 times, and the provided metric is computed as the average of the 3 experiments with lower execution time. The input data, $R$ and all the queries, are stored in DRAM just before starting execution. The Likwid tool [12] is used for measuring H/W performance counters: execution cycles, instructions executed, and number of Bytes read and write from DRAM. A Likwid command cleans up the NUMA domain at execution start to assure the same behavior in each execution.

Overall performance is computed as the average CPU time for each query ($\text{time}/\text{query}$), and the amount of Bytes read from memory is also averaged per query ($\text{Bytes}/\text{Query}$). Other metrics are the instructions executed per query ($\text{Icount}/\text{query}$), IPC (instructions per cycle) per execution core, and total DRAM bandwidth consumption.

We use OpenMP to statically distribute among threads the execution of the outer for loop, which iterates on the 10 million input queries. Data prefetching instructions are used on each search step to shift the memory request in time respect to the point when the data is actually needed, increasing the overlap between memory access and computation and reducing memory waiting time [13].

4.2. Computational Analysis

The 1-step FM-index inner loop consists of a recurrence of dependent memory loads and counting operations (read stage followed by compute stage, followed by read stage ...). Basically, each iteration: (1) reads two $F$ entries at addresses calculated from the interval ($\text{low}$, $\text{high}$), and then (2) uses the entries content to count symbols and generate the next interval. Furthermore, due to the pseudo-random nature of the input queries and the characteristics of BWT, memory accesses will be spread along the whole $F$ structure with a lack of temporal locality. For large references, the consequence is that most of the accesses will miss the LLC and data must be read from DRAM. The lack of memory-level parallelism on the inner loop exposes those large DRAM latencies in the execution time. Prefetching instructions are used to overlap part of the DRAM latency with computation.

There is plenty of potential parallelism when considering independent query searches, which can be exploited in the form of Thread-level Parallelism (TLP) to improve memory bandwidth utilization. Multiple threads generating independent memory requests may fulfill the memory pipeline to better exploit the available bandwidth. However, FM-index accesses exhibit very low spatial locality: only a 4-Byte counter ($LF$) and an 8-Byte bitmap
(BMP) are read on each memory read stage. Our implementation groups counters and bitmaps together in a single contiguous entry (see Fig. 3.b) to maximize spatial locality, but it is not enough to fully exploit DRAM bandwidth.

The \( n \)-step algorithm collapses \( n \) iterations into a single one. This scheme reduces the total instruction count and, in a lower order, the amount of data read by the program. The effect on the instruction count is similar to that produced by loop-unrolling: fewer loop control instructions and folding up some operations. Additionally, slightly less data is requested by the program because a single \( LF \) counter replaces \( nLF \) counters, but the total amount of bitmap information read by the program is basically the same regardless of \( n \).

In any case, the most important benefit of the \( n \)-step scheme is to aggregate the data required for advancing the search step by \( n \) symbols. The overall effect is an increase of spatial locality and a better exploitation of the available DRAM bandwidth. In brief: we replace many pseudo-random, dependent memory requests to scattered small data items by fewer requests to larger data items. Our experiments will support this analysis.

4.3. Performance of the \( n \)-step FM-Index search

Fig. 4.a compares performance for varying reference sizes, \(|R|\), and steps, \( n \). Increasing \( n \) almost always provides better performance in the range analyzed. There is a sharp performance degradation when \(|R|\) reaches values on the order of 10M as the corresponding index does not fit into the LLC. It is after this point when our proposal provides the higher benefits, achieving speedups between 1.4\( \times \) and 2.4\( \times \) (see Fig. 4.b). As stated in the introduction, interesting sequence alignment problems handle very large references.

![Query search time and SpeedUp due to increasing n](image)

Fig. 4. (a) Performance (ns/query) when varying \( n \) and \(|R|\); (b) Speedup when increasing \( n \) for selected values of \(|R|\).

4.4. Efficiency of sequential and parallel execution

As expected, the instruction count per query (\( Icount/Query \)) is almost constant for growing \(|R|\). Results for \( n = \{1, 2, 3, 4\} \) are \( Icount/Query = \{10K, 7.7K, 7.4K, 7.2K\} \), with a 23\% reduction for \( n = 2 \), and around 27\% for larger \( n \)’s. Fig. 5.a shows the IPC per core for varying \( n \) and \(|R|\). Small values of \(|R|\) allow achieving IPC’s between 2 and 2.5, fairly near to the theoretical peak of 3-4. Large values of \(|R|\) provoke LLC misses that reduce IPC between 3 and 5 times. The effect of large DRAM latencies is clearly seen here and performance becomes memory-bounded. Also, increasing \( n \) provides higher IPC’s, meaning a better tolerance to memory problems.

Fig. 5.b shows the good scalability of our implementation of FM-index when exploiting TLP. Using more execution cores provides almost linear speedup. Exploiting the H/W multithreading capability of each core still improves performance by \( \{1.67x, 1.68x, 1.56x, 1.49x\} \) for growing \( n \), which proves that single-thread performance is limited by memory dependences and latencies, as expected.

Multithreading, memory prefetching, and our \( n \)-step proposal have been effective to hide large DRAM latencies, but the best performing configuration still works with IPCs 3 times lower than the case where indexes fit into LLC. Performance for large indexes is still bounded by memory bandwidth issues.
4.5. Efficiency of Memory operations

Fig. 6.a compares the amount of data requested by the application (left) with the amount of data actually read from DRAM (right). The ratio requested/read for \( n = \{1, 2, 3, 4\} \) is \( \{0.16, 0.24, 0.28, 0.28\} \), indicating a very low efficiency of the memory hierarchy. Our proposal, which increases the spatial locality of data accesses, improves DRAM access efficiency by \( 1.5 \times -1.8 \times \). Results also show a \( 1.2 \times -1.4 \times \) reduction of data read requirements when increasing \( n \). These two factors provide the major benefits of our proposal.

4.6. Trading Memory requirements for Performance

The cost of \( n \)-step FM-index is an increase in memory storage requirements. We can reduce index size using a larger distance \( d \), which increments computational work and the total amount of data read from memory. We have evaluated an initial version of the algorithm for large values of \( d \). This version is based on the highly-optimized implementation for \( d = 32 \), and, for example, does not fully exploit the extra data-level parallelism provided by larger \( d \)'s (by means of SIMD instructions).
Fig. 7 shows how performance degrades as we increase $d$. Compression is more effective for large values of $n$, since it reduces the number of counters, which grow exponentially with $n$. However, at certain point, increasing $d$ provides lower compression ratios at the cost of higher performance penalties. The case example depicted in Fig. 7 shows that when DRAM size is very restrictive (less than 1.2 GB) the best solution is always the single-step method with an appropriate value of $d$. However, if we have additional DRAM space, we can use it to improve performance. With 2 GB, we may improve performance by $1.2\times$ by using a 2-step FM-index with $d = 128$. Doubling DRAM size to 4 GB provides an additional $1.5\times$ improvement when reducing $d$ to 32. Additional DRAM capacity does improve performance, but with diminishing returns.

5. Related Work

Ferragina et al. [14] introduced a methodology to evaluate the performance of a collection of compressed, full text, self-index, publicly available algorithms on a given collection of data sets. A similar methodology presented by our work was introduced in Salavert [15]. In this case, authors describe several GPU optimization parameters for BWT-based short read mapping to minimize I/O overheads for large query datasets.

All short read alignment applications used in major sequencing laboratories like Bowtie[1], SOAP[2], BWA[3], and GEM[4] do include multithread implementations to make use of current multicore CPUs. Common technique used is to distribute reads to be aligned among concurrent search threads. Published performance results are for the overall application, using the FM-index algorithm as a way to implement inexact matching. Our work provides isolated and more detailed insight on the performance of the core search algorithm.

The impact of pseudo-random dependent memory accesses on poor main memory performance was well described in [16]. Authors state the difficulty of predicting those accesses from the CPU and their lack of locality.

Recently, usage of multicore CPUs with additional GPU computing power has become a search hot topic. Recents contributions like Barracuda [17] and Cudshaw [18] describe the performance of inexact short read mapping using BWT-indexes. Both references state open challenges when analysing the performance of multicore implementations to overcome the poor data locality shown by the frequent thread accesses to memory.

Finally, alternative algorithmic strategies to optimize memory access patterns have been presented for the case of the tree structured index in [19].

6. Conclusions and future work

In this work we have analysed the performance of FM-index based exact match, the core algorithm used for aligning short sequences to a large genomic reference. We have presented a methodology for analysing the computational behaviour of the algorithm when running on a multicore CPU. Using the methodology, we have find the one-step FM-index search to perform poorly due to its pseudo-random access pattern and its low spatial locality. We present $n$-step FM-index algorithm that, combined with multithreading and memory prefetching
techniques proved to be effective to better exploit the available memory bandwidth. Improvements found show an improvement range from $1.4 \times$ to $2.4 \times$ versus the original FM-index search algorithm.

The $n$-step FM-index algorithm provides higher performance at the expense of higher memory capacity requirements. This tradeoff is alleviated by the reduction of DRAM cost. Our work prioritizes the goal of making a better use of the memory bandwidth of current computer technology in front of the minor objective of reducing the memory footprint of the index. Finally, we provide some principles to help evaluate which would be the best index configuration to use for a given amount of DRAM available in the target computer system.

Performance improvement in current memories is mostly based on the increment of DRAM burst and cache line sizes. In the next computer systems we are expecting to see that, in terms of efficiency, pseudo-random memory access algorithms will steadily lag behind sequential access algorithms. That is because the former algorithms cannot make good use of the technological improvements brought by new memories. In this case, to effectively improve algorithm efficiency we need to apply new alternative design strategies so that memory access patterns become more sequential. Our proposal represents an example in that direction.

Next steps in our research will be exploiting the exposed parallelism of the $n$-step FM-index in the form of wider SIMD/vector parallelism. This technique relies on finding data-level parallelism in the application. In the case of our algorithm, DLP can be found when increasing the distance among counters, and it can be used to alleviate the performance degradation when the index is compressed. This is of special interest for the case of accelerator architectures like GPUs and the recent intel Phi processor. We are also interested in reducing the memory footprint of the index by applying additional compression strategies. In any case, it will be important to evaluate the scalability of inexact search algorithms based on the indexes and search strategies defined.

7. Acknowledgements

This research has been supported by MICINN-Spain under contract TIN2011-28689-C02-01.

References