Proyecto Final de Carrera

Análisis y optimización de GEM: Una librería para el análisis e indexación de información genética

Santiago Marco Sola

Curso 2009/2010

Director

Jorge Albericio Latorre

Ponente

Pablo Ibáñez Marín

Grupo de Arquitectura de Computadores de Zaragoza (gaZ)
Departamento de Ingeniería e Informática de Sistemas (DIIS)
Centro Politécnico Superior (CPS)

Universidad de Zaragoza (UZ)

A mis amigos de Huesca, por estar siempre allí.

A mis amigos de Zaragoza, por el apoyo y los ánimos.

A mis tutores Victor Viñals y Pablo Ibánez, por darme esta maravillosa oportunidad.

A mi director, Jorge Albericio, por la dedicación y paciencia que ha tenido conmigo.

A Daniel Larraz, por ser la única persona que ha estado a la altura de mis locuras.

A Javier Campos, por encontrarme y ayudarme cada vez que he estado perdido.
Gracias por creer en mí.

A mi hermana, porque no puedo pensar en una forma mejor de seguir que siguiendo tus pasos.

A Mayel, por ser tan buena conmigo y entenderme tan bien como lo haces.

A mi madre, único testigo de todo el esfuerzo
dedicado en esta carrera, de los madrugones y las noches largas.
Gracias por nunca dejar de creer en mí y por ser el apoyo
y la cordura que han hecho posible que me sacara la carrera.

Esto va por ti.

Resumen

La librería GEM, que utiliza la transformada de Burrows-Wheeler y los índices de Ferragina-Manzini, es utilizada por los centros de investigación genómica para indexar grandes cantidades de pequeñas secuencias de DNA. Esta librería proporciona un conjunto de operaciones para anlizar de forma eficiente las secuencias dentro de un índice genómico. Por ello, se busca maximizar el rendimiento de esta aplicación en el entorno de producción.

Este Proyecto Fin de Carrera consiste en analizar y evaluar la librería, sus estructuras y mecanismos de indexación. Se analiza su rendimiento y comportamiento en memoria prestando especial atención al uso que realiza de la jerarquía de memoria. Así bien, se muestra cuales son los cuellos de botella. Además, se plantean alternativas de implementación enfocadas a mejorar el rendimiento de la librería. Se proponen mejoras tanto a nivel algoritmo como consientes de la arquitectura.

Una vez expuesto el análisis sobre la librería se exponen los resultados derivados de la implementación de las optimizaciones. Se muestran los resultados de ajustar los parámetros de optimización, los costes y resultados. De este modo, se analiza desde una perspectiva cualitativa y cuantitativa el impacto de las optimizaciones en la librería y porque ayudan a mejorar el rendimiento global de la librería.

Por otro lado, se exponen los resultados de varios estudios relacionados con el impacto de las opciones de compilación en la librería, la organización a bajo nivel del índice en memoria, la distribución de las bases en el índice y la implementación de operaciones en el camino crítico de la aplicación. Por último, se realiza una aproximación a una versión paralela de la librería. Esta ha sido implementada y evaluada en términos de rendimiento y escalabilidad. Se justifica la solución adoptada y los resultados obtenidos.

Se finaliza haciendo una evaluación del trabajo realizado y el planteamiento de objetivos en la línea del presente Proyecto Fin de Carrera.

Contenido

1. Introducción 9
1.1 Motivación y contexto 9
1.2 Objetivos del proyecto 9
1.3 Trabajo Previo 10
1.4 Organización de la memoria 10
2. Descripción del método de indexación. FM index 11
2.1 Descripción general del método de indexación 11
2.2 BWT. Burrows-Wheeler's Transform 11
2.3 FM-index 12
2.4 Análisis del diseño del FM-Index en GEM 13
3. Caracterización de la librería GEM 15
3.1 Modelado del rendimiento 15
3.2 Uso de la jerarquía cache 16
3.3 Caracterización de los accesos a memoria 17
4. Análisis de optimizaciones de la librería GEM 18
4.1 Motivación y organización 18
4.2 Tunning fm_bsearch 18
4.2.1 Memoization 18
4.2.2 Eliminación de la doble llamada al kernel. Condición Hi-lo=1 19
4.2.3 Prefetch de búsquedas entrelazadas 20
4.2.4 Optimizaciones enfocadas a maximizar el rendimiento del compilador 21
4.2.5 Especialización del código y clonación del kernel 21
4.2.6 Optimizaciones conscientes de la arquitectura 21
4.3 Tunning fm_lookup 23
4.3.1 Modificación de la estrategia de marcado 23
4.4 Análisis de la versión final 24
5. Análisis de la versión paralela de GEM 26
5.1 Paralelización por división de secuencias de entrada de la versión inicial 26
5.2 Paralelización de la versión optimizada 27
6. Organización del trabajo 28
6.1 Metodología de trabajo 28
6.2 Calendario, gestión de horas y esfuerzos 29
7. Conclusiones 30
7.1 Resultados del proyecto 30
7.2 Líneas de trabajo futuro 31

7.3 Valoración personal 31

A. Introducción al contexto de la bioinformática 32

A.1 Reunión con Paolo Ribeca (CRG. Barcelona)...34

A.2 Reunión con Carmen Cons (Facultad Veterinaria. Zaragoza) 35

B. Análisis en detalle del rendimiento de la implementación 36

B1. Análisis del rendimiento para diferentes compiladores y opciones de compilación. 36

B1.1 Rendimiento con gcc 37

B1.2 Rendimiento con icc 40

B1.2 Comparativa gcc/icc 43

B2. Comportamiento para diferentes configuraciones de cadenas de búsqueda 44

B3. Análisis de los cuellos de botella 47

B3.1 Rendimiento de fm_bsearch..47

B3.2 Rendimiento de fm_lookup 47

B3.3 Informe del Profiler gprof 48

C. Análisis en detalle del comportamiento en memoria 51

C1. Simulaciones para el dimensionado de la cache 51

C2.Análisis de los accesos al índice y distribución de las cadenas en el mismo 55

C2.1 Gráficos de próximos accesos para la función fm_bsearch 55

C2.1.2 Gráficos de accesos al índice ordenados para la función fm_bsearch.....................................62

C2.1.3 Gráficos de accesos al índice por cadenas para fm_bsearch...65

C2.2 Análisis de los accesos a memoria de la función fm_lookup..68

C2.2.1 Gráfico de accesos al índice para la función fm_lookup 69

C2.2.2 Análisis de los anclajes para la función fm_lookup..70

D. Análisis de propuestas en la construcción de indices FM e implementación de kernels..73

D1. Propuestas a la organización a bajo nivel del índice de DNA..73

D2. Propuestas a la implementación de los kernels...74

D3. Propuestas a la implementación de la operación popCount 75

D4. Propuestas a la organización de CSA...78

D5. Dimensionado de la tabla de lookups para memoization 81

D6. Dimensionado del número de queries entrelazadas...82

E. Caracterización de la organización del índice genómico de referencia...............83

F. Aplicaciones para el análisis de la biblioteca y del índice................................87

G. Bibliografia 89

1. Introducción

The White Rabbit put on his spectacles.
"Where shall I begin, please your Majesty?" he asked.
"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."
- Lewis Carroll, Alice's Adventures in Wonderland

1.1 Motivación y contexto

Las nuevas plataformas de secuenciación son capaces de generar grandes cantidades de información genómica al día. Se estima que la cantidad de información genómica producida por estas máquinas en el 2012 excederá los 15 petabytes al año. Debido a ello se están desarrollando nuevas herramientas que puedan procesar esta enorme cantidad de información. Son de vital importancia librerías básicas de indexación, alineamiento o ensamblado de DNA.

El grupo de arquitectura de computadores de la UZ (gaZ) está colaborando en el proyecto Consolider "Supercomputacion y e-ciencia" dónde participan los principales grupos de arquitectura de computadores del país e importantes grupos de científicos que utilizan como herramienta habitual la plataforma de supercomputación Mare Nostrum del Centro Nacional de Supercomputación de Barcelona. En concreto, se ha iniciado una colaboración con el CRG (Centro de Regulación Genómica, Barcelona) y el recientemente creado Centro Nacional de Análisis Genómico (CNAG), consistente en el estudio del rendimiento y mejora de GEM: Un conjunto de algoritmos para analizar/indexar las grandes cantidades de información genética producidas por las citadas plataformas de secuenciación.

La librería GEM se basa en la utilización de la transformada de Burrows-Wheeler y los índices de Ferragina-Manzini para indexar de forma rápida grandes secuencias de DNA. Este método además proporciona un conjunto de operaciones que permiten analizar de forma eficiente estas secuencias dentro de un índice de referencia. Esta librería es la base de múltiples aplicaciones que necesitan del mapeo de DNA como operación fundamental.

1.2 Objetivos del proyecto

Este Proyecto Final de Carrera consiste en comprender y caracterizar la librería GEM, analizar los cuellos de botella y estudiar alternativas para optimizar su rendimiento. Se prestará especial énfasis a las alternativas que hagan un uso optimizado de la jerarquía de memoria y del paralelismo de grano fino ofrecido por las extensiones multimedia. Además se estudiará la paralelización del algoritmo y su impacto en el rendimiento y utilización de la memoria.

En concreto, las tareas a realizar son:

- Entender la naturaleza del problema y los diferentes contextos de aplicación.
- Comprender en profundidad las estructuras de datos y algoritmos de indexación utilizados.

- Estudiar la implementación de los índices Ferragina-Manzini basados en la transformada de Burrows-Wheeler dentro del código de GEM.
- Detectar los cuellos de botella, centrándose en el paralelismo de instrucción y en la jerarquía de memoria.
- Proponer soluciones para aumentar el rendimiento de la aplicación single-thread, aprovechando las posibilidades de la microarquitectura, su jerarquía de memoria y del compilador.
- Realizar un estudio preliminar sobre la posibilidad de explotar grados mayores de paralelismo, a través de la vectorización o el paralelismo de memoria compartida.

1.3 Trabajo Previo

Para la realización de este PFC se ha contado con la versión pre-release de GEM. Paolo Ribeca, el autor de esta librería, nos proporcionó los ficheros fuentes básicos de esta librería. Además nos proporciono el índice FM del genoma humano (índice de referencia de ahora en adelante). En base a estos ficheros, se ha analizado el rendimiento de esta primera implementación y se han propuesto optimizaciones sobre la misma.

1.4 Organización de la memoria

En el apartado 2. Caracterización del método de indexación ,se realiza una breve introducción al método de indexación basado en los índices FM y la transformada Burrows-Wheeler. Además se explica el diseño en la librería GEM, las estructuras que utiliza y las funciones que implementa.

En el apartado 3. Caracterización de la librería GEM, se expone un resumen sobre el análisis realizado del rendimiento de la librería. Además se expone el uso que hace de la jerarquía de memoria y se exponen datos cuantitativos del comportamiento en memoria.

En el apartado 4. Análisis de optimizaciones de la librería GEM, se exponen las alternativas de implementación dirigidas a mejorar el tiempo de ejecución. Para cada una de ellas se expone la causa que fundamenta su implementación y los resultados que de ella se derivan.

En el apartado 5. Análisis de la versión paralela de GEM, se exponen los resultados obtenidos de la paralelización de la librería.

En el apartado 6. Organización del trabajo, se ofrece un resumen sobre la metodología de trabajo, la planificación seguida y las hora invertidas en este proyecto.

En el apartado 7. Conclusiones, se exponen las conclusiones acerca del proyecto y realiza un breve comentario sobre las futuras líneas de trabajo junto con una breve valoración personal.

2. Descripción del método de indexación. FM index

"The time has come," the Walrus said,
"To talk of many things:
Of shoes and ships and sealing wax,
Of cabbages and kings.
And why the sea is boiling hot.
And whether pigs have wings."
- Lewis Carroll, Through the Looking-Glass

2.1 Descripción general del método de indexación

Ferragina y Manzini proponen en [1,2] un índice comprimido auto-contenido (*Compressed self-index*) para la indexación de textos. Un *self-index* es una clase de índice que además contiene el propio texto que indexa sin necesidad de almacenarlo separadamente. El diseño del FM-index se basa en la utilización de la transformada de Burrows-Wheeler[3] junto a una serie de métodos de compresión y un *Suffix-Array* comprimido. De esta forma, la estructura resultante forma un *Suffix-Array* comprimido con una compresión que está cerca del mínimo teórico dado por la entropía de orden cero[4]. Es decir, con un *FM-index* podemos indexar un texto comprimido sin necesidad de tener almacenado el texto original[5,6,7].

Este método constituye un referente en lo que a índices comprimidos se refiere. Sobre la propuesta inicial se han realizado un conjunto bastante amplio de propuestas que introducen variaciones en los métodos de compresión utilizados [9], la estructura del CSA[1] [2] y algunos otros ámbitos del método [10,11].

2.2 BWT. Burrows-Wheeler's Transform

Burrows y Wheeler introdujeron un método de compresión basado en la transformada que ahora se conoce como BWT [12]. Esta transformada consta de tres pasos. En el primero, se añade al final de nuestro texto T un carácter especial para marcar el final del mismo #. El segundo paso es formar una matriz conceptual M_T donde las filas son las rotaciones cíclicas de T# ordenadas de forma lexicográfica. Por último, cogemos la última columna de M_T, que denominaremos T^{bwt} y constituirá nuestro texto transformado. Podemos ver gráficamente este proceso en la figura 2.1.

Observar que cada columna de la matriz M_T es una permutación de T#. Por ello se puede ver que la primera fila F es la ordenación de los caracteres de la cadena T#.

[1] CSA. Compressed Suffix Array

```
                           F              T^bwt
mississippi#               # mississipp i
ississippi#m               i #mississip p
ssissippi#mi               i ppi#missis s
sissippi#mis               i ssippi#mis s
issippi#miss               i ssissippi# m
ssippi#missi      ==>      m ississippi #
sippi#missis               p i#mississi p
ippi#mississ               p pi#mississ i
ppi#mississi               s ippi#missi s
pi#mississip               s issippi#mi s
i#mississipp               s sippi#miss i
#mississippi               s sissippi#m i
```

Figura 2.1 Transformada BWT

En sí, el BWT no es un método de compresión, solo una transformación del texto que, como veremos más adelante, es reversible. No obstante, al ser T^{bwt} la columna anterior a F (con F ordenada lexicográficamente), T^{bwt} contiene reflejo de esa ordenación en forma de secuencias (*runs*) de caracteres iguales. Esta propiedad hace del texto muy adecuado para métodos de compresión como *run-length*. Ahora, dado que hemos ordenado las permutaciones del texto T#, hemos construido de alguna forma un vector de sufijos. Para mostrar de forma clara esta relación, definimos un conjunto de estructuras y operadores.

- Sea C[·] un vector de longitud el tamaño de nuestro alfabeto ($|\Sigma|$) donde C[c] contiene el número de caracteres del alfabeto lexicográficamente menores que c.
- Sea Occ(c,q) el número de ocurrencias del carácter c en el prefijo T^{bwt} [1,q].
- Sea LF(i) = C[T^{bwt} [i]] + Occ(T^{bwt} [i],i) la función que nos permite recorrer el texto hacia atrás (*backward scanning*). Con LF podemos recuperar el carácter previo a uno dado cualquier en $T^{bwt.}$

Se puede demostrar como desde la posición 1 del T^{bwt} podemos aplicar sucesivamente la función LF (*last-to-first*) e ir recuperando el texto original desde la posición del finalizador #.

2.3 FM-index

El índice FM es una forma de T^{bwt} comprimido junto con un CSA (*compressed suffix array*) que permiten calcular la función Occ (rango de un carácter) en tiempo constante O(1). El FM-index se compone de dos funciones fundamentales. Por similitud con la implementación las llamaremos fm_bsearch y fm_lookup.

La función fm_bsearch se encarga de buscar un sufijo dado en el índice. Mediante llamadas a Occ, los centinelas First y Last (o Hi y Lo en la implementación) acotan la porción de índice FM que contiene sufijos cuyo postfijo es P[i,p]. Es decir, la iteración i-ésima del bucle cumple el siguiente invariante: *First apunta a la primera fila de la Matriz M_T que tiene como prefijo P[i,p] y Last a la última fila de la Matriz M_T que tiene como prefijo P[i,p].* Se puede demostrar que, según lo dicho anteriormente, Last-First+1 nos da el conteo total de sufijos encontrados que coinciden con la cadena buscada.

```
Algoritmo FM_Search(P, T^bwt)
    i = |P|;
    first = 1; last = |T^bwt|;
    while ((first <= last) and (i >= 1)) do
        c = P[i];
        first = C[c] + Occ(T^bwt,c,first-1) + 1;
        last = C[c] + Occ(T^bwt,c,last);
        i = i - 1;
    endDo
    if ( last < first ) then return "no encontrado"
    else return "encontradas (last-first+1) ocurrencias"
```

Figura 2.2 Algoritmo de búsqueda hacia atrás

No obstante, las posiciones devueltas entre First y Last no corresponden al texto original, sino al T^{bwt}. Por ello, debemos decodificar estas posiciones. Para ello utilizaremos la función fm_lookup. Como ya se ha comentado, junto a T^{bwt} disponemos de un CSA que mantiene una correlación entre posiciones del índice FM y el texto original. No obstante, no almacena todas las posiciones posibles, sino que almacena una posición cada cierto intervalo de sufijos. Denominaremos a estas posiciones almacenadas de forma explícita *anclajes*. De este modo, supondremos que los anclajes están distribuidos de forma homogénea con respecto al texto original.

La idea es recorrer el texto original hacia atrás hasta encontrar una posición del texto cuya decodificación halla sido almacenada explícitamente en el CSA. De esta forma la posición buscada se puede expresar como la suma de la posición recuperada y la distancia entre la posición buscada y la posición almacenada.

2.4 Análisis del diseño del FM-Index en GEM

Tanto el algoritmo de búsqueda hacia atrás (fm_bsearch) como el de decodificación de posiciones (fm_lookup) dependen de la función LF. El tiempo de ejecución está dominado por el cálculo de la función Occ. Para que este cálculo se pueda resolver en tiempo O(1), se precalcula y almacena el conteo los cuatro caracteres {A,C,G,T} para ciertas posiciones del índice. De este modo, se aplica una

estructura de doble bucketing al índice para el cálculo de Occ[2]. La estructura concreta utilizada en la implementación de GEM se muestra en la figura 2.3.

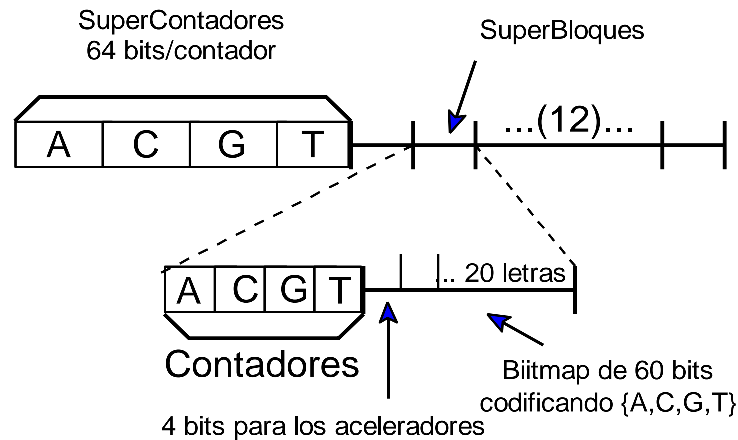

Figura 2.2 Estructura del índice

Cada contador de superbloque almacena el conteo de su correspondiente carácter hasta esa posición. Y el contador de bloque almacena el conteo desde el último superbloque hasta esa posición. Los caracteres individuales están codificados en el bitmap en orden. Notar que la implementación actual no aplica ninguna técnica de compresión ni sobre el bitmap ni sobre los contadores. Solo emplea una agrupación compacta de la información.

Dada esta estructuración, para el cálculo de Occ dadas un posición y un carácter, se han de sumar el contador de bloque y superbloque de ese carácter y el número de ocurrencias del mismo en el bitmap correspondiente a esa posición. Con esto se consigue un tiempo constante en el cálculo de Occ.

Por último, mencionar que la estructura del CSA es muy sencilla ya que solo está formado por las posiciones decodificadas dispuestas en orden secuencial. Es responsabilidad del algoritmo fm_lookup encontrar que posición del CSA corresponde con la posición dada para decodificar.

[2] Es por esta organización que el índice FM es una estructura de datos compacta

3. Caracterización de la librería GEM

All this time the Guard was looking at her, first through a telescope, then through a microscope, and then through an opera- glass. At last he said, `You're travelling the wrong way,' and shut up the window and went away. - Lewis Carroll, Through the Looking-Glass

3.1 Modelado del rendimiento

El primer análisis que se realizo consistió en implementar un sencillo indexador[3] que se ejecutará sobre un conjunto de prueba y utilizar el perfilador gprof para analizar el rendimiento. Como resultado se muestra la figura 3.1 con un resumen de las funciones que más tiempo de ejecución consumen.

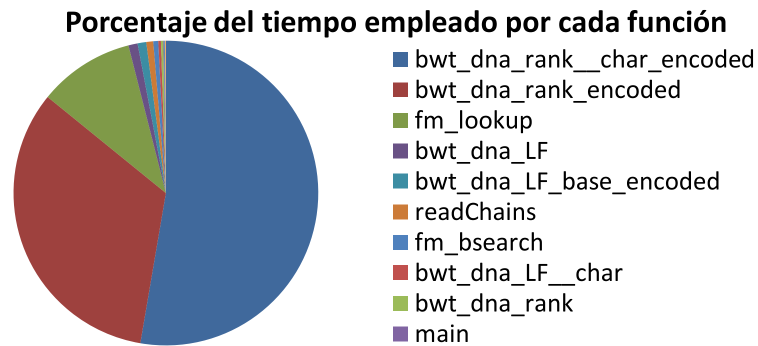

Figura 3.1. Porcentaje de tiempo empleado por cada función

Podemos ver como más del 80% del tiempo de ejecución es consumido por las funciones de acceso al índice[4] (para el cálculo de Occ). Notar también que fm_lookup tiene un peso relevante dado que, además de acceder al índice, se encarga de acceder al CSA. Por otro lado, el la figura 3.2 podemos ver como estas funciones son llamadas múltiples veces a lo largo de la ejecución del programa[5]. Por ello, nuestro análisis se centra en optimizar este cuello de botella. Por un lado reducir el número de llamadas al kernel (en la medida que el algoritmo lo permita) y por otro reducir el coste de cada llamada.

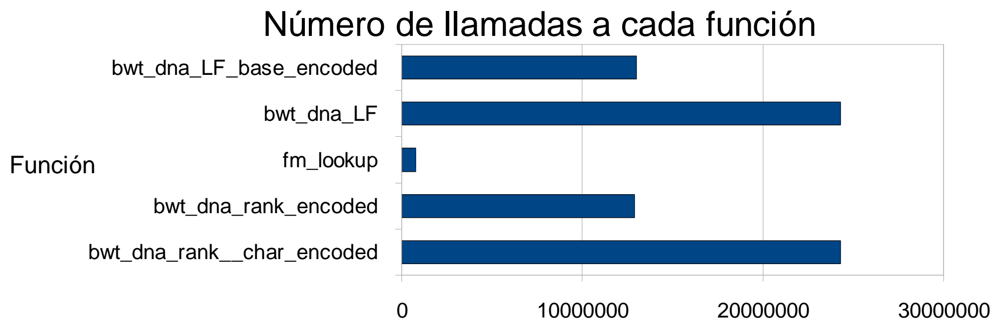

Figura 3.2 Número de llamadas a cada función

[3] Un sencillo indexador es una aplicación que indexa un juego de cadenas de entrada dados en el índice FM de referencia. Ver apartado 6.1 Metodología

[4] Kernels

[5] Son llamadas dos veces por cada búsqueda de un carácter de DNA del juego de pruebas

3.2 Uso de la jerarquía cache

Con el objetivo de descubrir la causa subyacente del coste en las llamadas al kernel del programa se simuló con cacheGrind el mismo programa. CacheGrind[13] es un simulador que permite configurar parámetros de la arquitectura como los tamaños de las cache y recoger datos como los fallos en la misma, fallos del predictor de saltos, etc. En la figura 3.3 podemos ver el número de fallos en L1/L2 por instrucciones ejecutadas y por accesos de memoria (respectivamente).

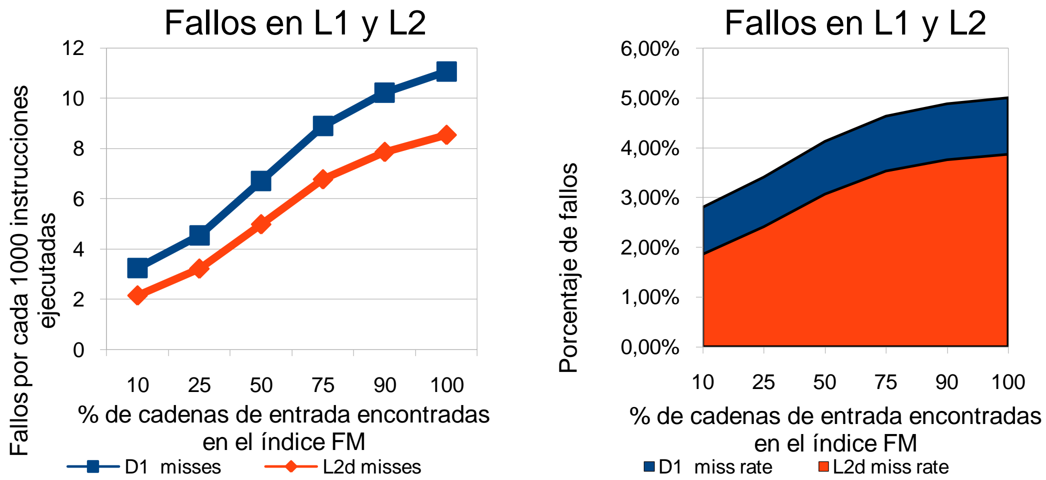

Figura 3.3 Fallos en L1/L2

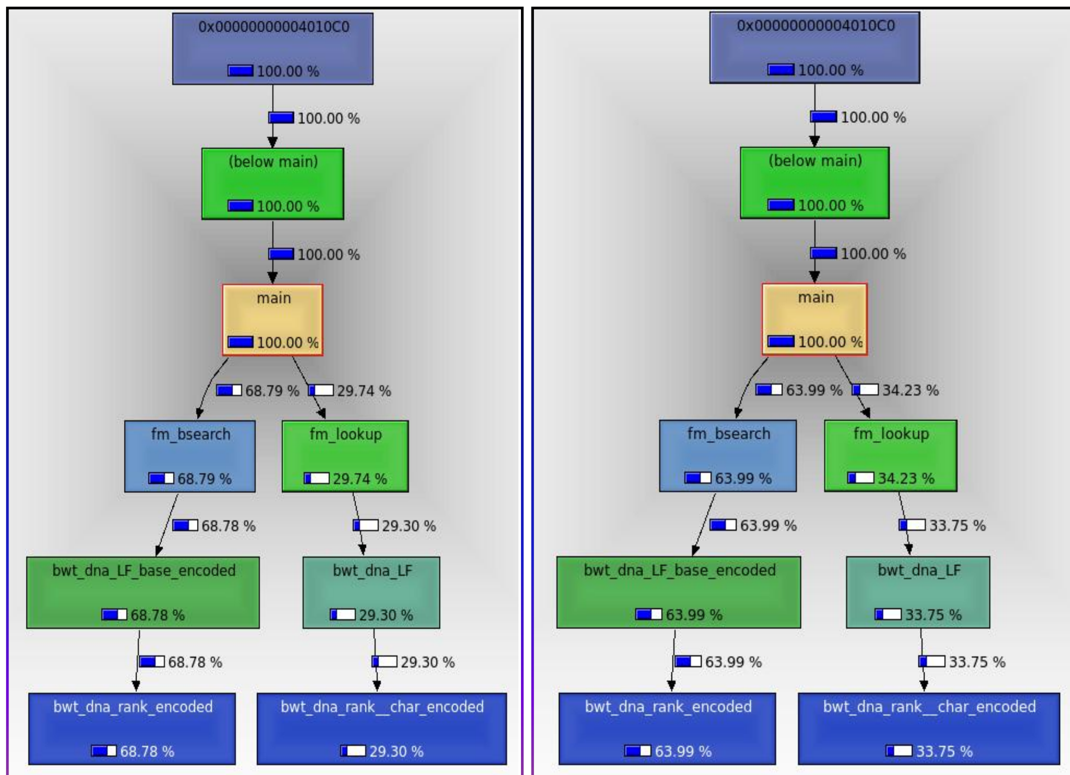

Figura 3.4 Árbol jerárquico de porcentaje de fallos en L1 y L2 respectivamente

Podría parecer que un 5% es un porcentaje muy bajo de fallos (obligatorios), no obstante hay que tener en cuenta que el programa incluye la carga del índice en memoria y que por tanto solo menos de un 6% de los accesos son debidos realmente a la búsqueda y decodificación. Por lo que el programa falla en L1/L2 cache en casi todos los accesos que realiza al índice. Además, podemos ver como, según se incrementa el porcentaje de cadenas contenidas en el índice del juego de pruebas, se incrementa el número de fallos en cache. Es decir, a mayor número de iteraciones del método, más accesos al índice se realizan y por tanto mayor número de fallos en cache se producen. De forma esquemática, en la figura 3.4 podemos ver como entre las funciones *bwt_dna_rank_encoded* y *bwt_dna_rank__char_encoded* suman el mayor porcentaje de fallos en L1 y L2.

3.3 Caracterización de los accesos a memoria

En un análisis más fino, se ha empleado el analizador de rendimiento Vtune[14] para tomar muestras precisas de eventos hardware en la plataforma Intel. Vtune es una aplicación que toma muestras de los contadores hardware instalados en los procesadores Intel y que registran eventos como fallos en cache, fallos en TLB, instrucciones ejecutadas, etc. A continuación, a modo de resumen, se muestra los datos más relevantes obtenidos.

	Accesos L1	Fallos L1	Fallos L2
GEMv0::bwt_dna_rank_encoded	2,85	1,77	1,48
GEMv0::bwt_dna_rank__char_encoded	25,24	5,78	3,06

Tabla 3.1 Resumen de Accesos y fallos por acceso al índice FM

	L2miss por 1000 Inst	L1miss por 1000 Inst	Stalls por Inst	TLBmiss
GEMv0::bwt_dna_rank_encoded	13,472	16,12	2,452	368
GEMv0::bwt_dna_rank__char_encoded	35,230	66,57	2,226	656

Tabla 3.2 Resumen de magnitudes relacionadas con el uso de la jerarquía de memoria por acceso al índice

Podemos ver como por cada acceso al índice que se realiza se producen un elevado número de fallos en L1 que, a su vez, se convierten en fallos en L2. Esto deteriora gravemente el rendimiento de la aplicación. Podemos ver como el índice de fallos en L2 por cada 1000 instrucciones es muy elevado. Además, por cada instrucción ejecutada se produce, en media, dos detenciones del pipeline. Y lo que es peor, debido a todas las ejecuciones de los kernel se producen más de 1000 fallos de página.

4. Análisis de optimizaciones de la librería GEM

> ***"Well, in our country," said Alice, still panting a little, "you'd generally get to somewhere else -- if you ran very fast for a long time, as we've been doing." "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"***
> - Lewis Carroll, Through the Looking-Glass

4.1 Motivación y organización

El conjunto de optimizaciones propuestas derivan de los resultados de la caracterización de la librería. Su objetivo es el de minimizar el impacto de algún factor o varios en la función de coste. Por ello, en primer lugar se muestra para fm_bsearch y fm_lookup la función analítica (figuras 4.1 y 4.6). A continuación, se presentan las distintas optimizaciones. Estas están ordenadas por nivel de abstracción; desde el nivel de sistema pasando por aplicación hasta el nivel de arquitectura.

4.2 Tunning fm_bsearch

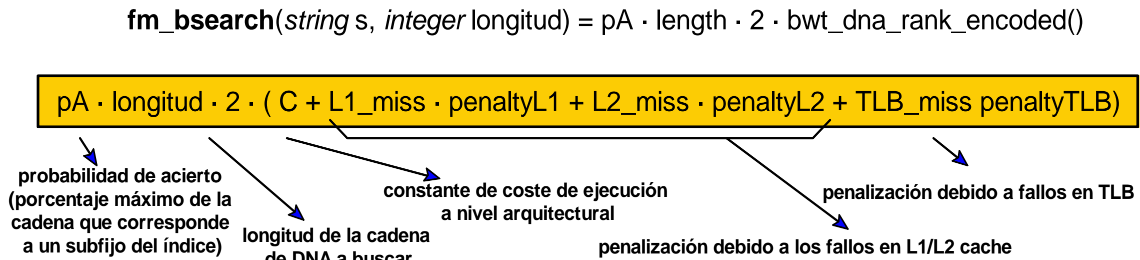

Figura 4.1 Función de coste de fm_bsearch

4.2.1 Memoization

Memoization es una técnica de optimización que se emplea cuando en un programa se realizan llamadas a funciones que calculan resultados ya procesados. En el caso de fm_bsearch, los centinelas hi y lo se inicializan siempre a los mismos valores. Por ello, los primeros siguientes valores de hi y lo ante la búsqueda de {A,C,G,T} se pueden almacenar en una tabla. De esta forma, los primeros n pasos de todas las búsquedas estarán precalculados. Esto ahorra los accesos al índice de los primeros pasos de la búsqueda con un coste extra de memoria.

De las alternativas planteadas[6], la creación de una tabla de lookup en la carga del índice es la que mejores resultados ha devuelto. Como se puede ver en la tabla 4.1, según se incrementa el tamaño de la tabla se reduce el tiempo total de ejecución. Por otro lado, también se puede ver que el incremento en espacio de la tabla es exponencial $O(|\Sigma|^n)$.

Dimension Tabla Lookup (4^n)	4	6	8	10	12	14
SpeedUp	1,022	1,040	1,095	1,157	1,225	1,286
Tamaño Memoria	256B	16KB	256KB	4MB	64MB	1024MB

Tabla 4.1 Speedup y coste para diferentes tamaños de tabla de lookup

4.2.2 Eliminación de la doble llamada al kernel. Condición Hi-lo=1

Del análisis del método de búsqueda en el índice FM se puede derivar la condición Hi-Lo=1. Cuando los dos centinelas (first y last) solo acotan un sufijo podemos afirmar que la búsqueda devolverá como mucho 1 resultado. Es decir, potencialmente solo existe 1 resultado para la búsqueda a partir de que hi-lo=1. De este razonamiento se puede derivar que una vez alcanzada esta condición no es necesario seguir acotando el intervalo de sufijos. En vez de eso, bastará con averiguar si el único sufijo potencialmente solución es realmente una solución.

No obstante la razón para aprovechar esta condición es el número tan elevado de veces que se presenta. Podemos ver en la figura 4.2 como en media un 80% de las búsquedas alcanzan esta condición. Y además, en la figura 4.3 se muestra como se alcanza esta condición antes de la iteración 20. Lo cual significa que entre 50-80 de las iteraciones realizadas pueden ser evitadas.

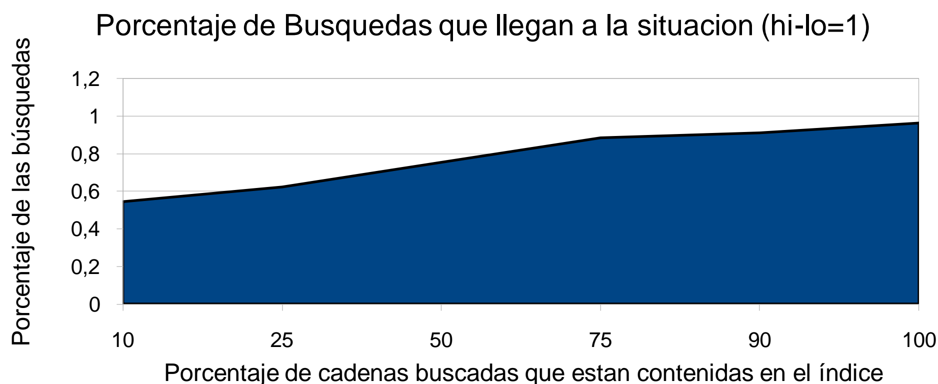

Figura 4.2 Porcentaje de búsquedas que llegan a la situación hi-lo=1

[6] ANEXO D5. Dimensionado de la tabla de lookups para memoization

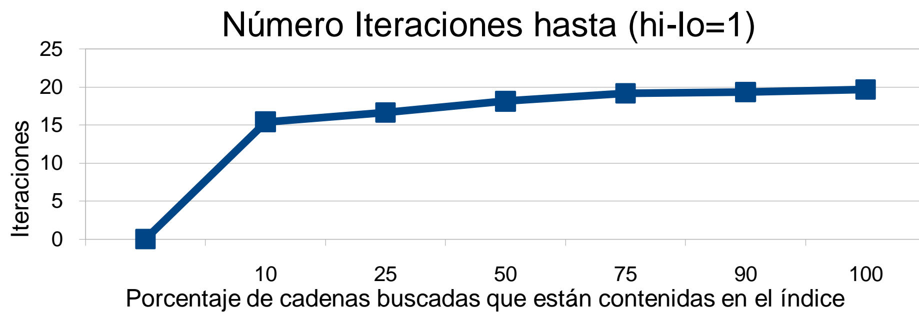

Figura 4.3 Número de iteraciones del bucle principal hasta hi-lo=1

Así en la versión final, se detecta esta condición y en los siguientes pasos se comprueba que el carácter a buscar concuerda con el carácter residente en la posición actual. En caso afirmativo se continúa buscando, en caso contrario la búsqueda finaliza.

4.2.3 Prefetch de búsquedas entrelazadas

Uno de los principales problemas de rendimiento de la librería son debidos al elevado número de fallos en L1 y L2 cache. En general, cada acceso al índice produce un fallo en L1 cache que se eleva a fallo en L2 cache. Esto es debido a que GEM es una librería intensiva en memoria. Con un índice de tamaño 2,7 GB (como el de referencia[7]) y los accesos pseudo-aleatorios del algoritmo el programa esta abocado a fallar en cache.

Como solución a este problema se plantea dividir las operaciones de acceso al índice en dos fases: fetch y retrieve. La ejecución de varias búsquedas se agrupará en paquetes de n de forma que todas progresaran a la vez. La dinámica de búsqueda consistirá en realizar una operación de fetch para todas las búsquedas del paquete incluyendo una llamada prefetch software para el bloque que se está buscando. Posteriormente, se realizará una operación retrieve para todas las búsquedas.

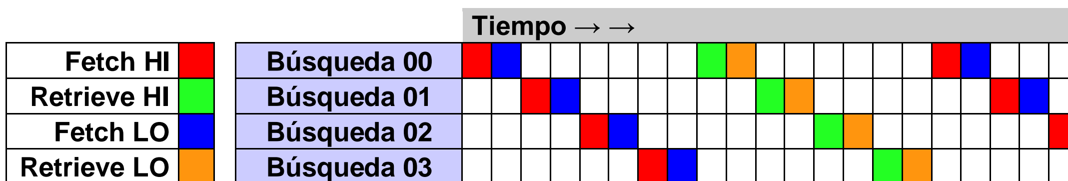

Figura 4.4 Prefetch de búsquedas entrelazadas

La idea principal consiste en que mientras se hace prefetch de una posición de memoria y el procesador sirve el bloque en la cache, se están procesando otras búsquedas. Para cuando le llega el turno a la búsqueda mencionada, su bloque ya está en cache. Por lo que no se produce fallo y el programa puede avanzar sin detenciones por fallos en cache. La figura 4.3 muestra de una manera gráfica como se entrelazan las búsquedas y como se producen ventanas de tiempo donde los bloques solicitados pueden llegar a la cache via instrucción prefetch software.

---

[7] Índice de referencia del genoma del Homo Sapiens. Ver apartado 6.1 Metodología

4.2.4 Optimizaciones enfocadas a maximizar el rendimiento del compilador

En la versión inicial de GEM las estructuras básicas se referenciaban como punteros a void. Una vez se necesitaba acceder a su contenido se realizaba un cast al tamaño de dato básico de la estructura. El objetivo era proporcionar funcionalidad a índices con direcciones de 32 y 64 bits. Se ha comprobado como esto afecta muy negativamente al compilador, ofuscando posibles optimizaciones derivadas de conocer de antemano el tipo de dato. Por ello, se ha planteado especializar el código para estos dos casos y compilar los dos por separado.

Por otro lado se ha realizado un análisis de rendimiento con diferentes compiladores y diferentes opciones de compilación[8]. Se eliminaron las directivas register, para que el compilador pudiera alojar las variables a su discreción y se reescribieron las cabeceras eliminando las directivas extern. Todo esto permite al compilador de intel generar binarios especializados hasta 6% más potentes.

4.2.5 Especialización del código y clonación del kernel

Por otro lado, los kernels de la aplicación hacían uso de parámetros como cntr_bytes, cntr_mask y period_bytes. No obstante, es raro que estos parámetros tomen valores fuera de un rango discreto y muy pequeño de valores. Por ello, se ha especializado el código para los pocos casos que se pueden dar en un entorno real de producción (que además son los que mayor rendimiento obtienen). Con solo esta optimización, se ha podido alcanzar hasta un 1,4x de speedUp.

4.2.6 Optimizaciones conscientes de la arquitectura

En primer lugar, se han corregido parámetros relacionados con la distribución del índice en memoria para que las funciones del kernel se redujeran en complejidad. A modo de ejemplo, se han escogido tamaños de superBloque y de bloque potencia de 2 para que las divisiones y multiplicaciones en los kernels queden simplificadas a desplazamientos (varios ciclos menos de ejecución). En la linea de nuevas estructuras de índice que beneficien a los kernels en ejecución se han propuesto varias alternativas y se ha estudiado su desempeño[9]. En última instancia se ha escogido la organización a bajo nivel que se presenta a continuación, figura 4.4.

---

[8] Anexo B1. Análisis del rendimiento para diferentes compiladores y opciones de compilación

[9] Anexo D. Análisis de propuestas en la construcción de índices FM e implementación de kernels

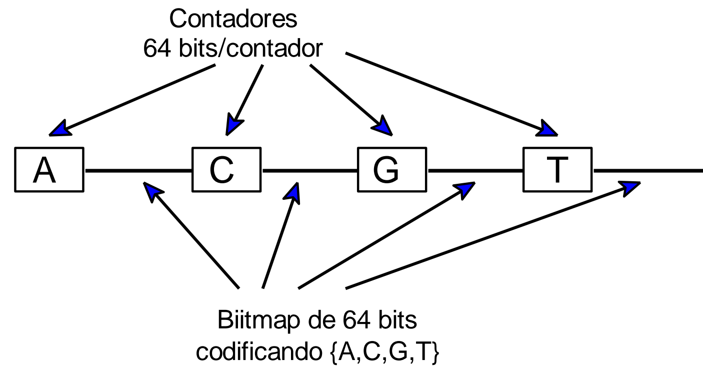

Figura 4.5 Estructura de la nueva organización del índice

Se han reducido los dos niveles de "bucketing" a uno solo. Con ello se realizan menos accesos al índice y localizados. Se reduce él numero de operaciones de localización del bloque del índice a costa de un incremento del tamaño del índice en un factor 1,5. Ademas, se han separado las codificaciones de los 4 caracteres en 4 bitmaps diferentes. Dado que en cada acceso solo se cuentan las ocurrencias de un carácter, ya no es necesario filtrar el carácter deseado con mascaras. Así, la operación de conteo final queda reducida a un "population count".

Ademas, se ha potenciado el efecto de los "aceleradores" cuya misión es testear si el popCount era necesario o por el contrario su resultado sera cero. De este modo, se ha escogido un tamaño de bitmap de solo 32 posiciones. Debido a las propiedades de la transformada BWT, las letras en el índice tienden a organizarse en largas secuencias del mismo carácter. Por ello, muchos bitmaps no contienen ningún bit a uno y esta organización potencia que se detecte a tiempo y no se tenga que ejecutar la operación de popCount. A raíz de esta idea se ha realizado un análisis de la estructuración del índice de referencia[10]. En este se puede ver de forma cuantitativa como la probabilidad de encontrar un bitmap a cero aumenta al reducir el tamaño. Y como por debajo de longitud 32 no merece la pena reducirlo (debido a que solo es significativo a partir de 8 y en este punto el índice ocupa 6,7GB).

Por otro lado, se ha realizado un estudio de diferentes alternativas a la implementación del popCount[11]. No obstante, siempre que el procesador sea un modelo posterior a un Nehalem Core i7 de Intel compatible con extensiones SSE4.2, se puede optar por utilizar la SIMD `_mm_popcnt_u32`. Esta intrínseca se ejecuta en una sola instrucción del procesador y se prescinde de inteligentes trucos con bits.

Por último, se ha observado el gran número de fallos en TLB que se producen y el impacto de estos en el rendimiento. Esto es lógico dado que utilizando el índice de referencia se direccionan 2,4 GB de memoria y el tamaño de página utilizado son 4KB. Por ello, se propone utilizar HugePages que permiten, en una arquitectura i386, páginas de 2MB. Esto reduce el número de páginas necesarias y con ello el número de fallos en TLB. Además tiene la ventaja añadida que estas páginas no son expulsadas de memoria.

[10] Anexo E. Caracterización de la organización del índice genómico de referencia
[11] Anexo D3. Propuestas a la implementación de la operación popCount

4.3 Tunning fm_lookup

Al compartir muchas similitudes, las optimizaciones aplicadas a los kernel de fm_bsearch son trasladadas a los kernels de fm_lookup. Por ello, también se utiliza una política de prefetch de búsquedas entrelazadas para gestionar la decodificación con fm_lookup. También se provechan las ventajas de la nueva organización del índice y ,en general, todas las optimizaciones conscientes de la arquitectura. En este apartado solo se mencionaran aquellas que aporten conceptos novedosos en el *tunning* de fm_lookup.

fm_lookup(*integer posicion*) = Dist · · bwt_dna_rank__char_encoded() + CSA_Access

Figura 4.6 Función de coste de fm_lookup

4.3.1 Modificación de la estrategia de marcado

La estrategia de elección de los anclajes (marcas) de la versión original estaba basada en el marcado equidistante del espacio del índice. Estas posiciones trasladadas al espacio de la transformada BWT se "barajan" de forma que no existe ninguna cota superior garantizada del número mínimo de pasos que la función fm_lookup ha de iterar. De este modo, se han obtenido resultados del número de iteraciones máximo que realiza el método (tabla 4.2).

Distancia entre anclajes	8	16	32	64	128
Numero de anclajes	357255743	178627872	89313936	44656968	22328484
Distancia media entre anclajes	7	15	30	63	126
Máxima Distancia	141	329	587	1092	2062
Minima Distancia	0	0	0	0	0
Tamaño del CSA (MB)	1362	681	340	170	85

Tabla 4.2 Cifras relacionadas con el comportamiento del CSA

Derivado de esta observación se han planteado alternativas a la elección de los anclajes en el índice[12]. Sin lugar a duda la mejor opción es la de el marcado inverso, que escoge los anclajes de manera equidistante en el espacio de direcciones de la BWT. Esto requiere de estructuras auxiliares para poder determinar las posiciones marcadas. No obstante, consigue acotar el número máximo de iteraciones del método.

[12] Anexo D4. Propuestas a la organización de CSA

4.4 Análisis de la versión final

De la propuesta de todas las optimizaciones se ha implementado una versión final que contenga todas las mejoras. Una vez comprobada la corrección de esta versión con varios juegos de pruebas se ha analizado su rendimiento para compararlo con la versión anterior. En primer lugar, se puede apreciar en la figura 4.7 como se han reducido los tiempos de ejecución totales. Se puede ver como la función que toma más tiempo en ejecución es readChains que se ocupa de la carga del juego de cadenas de DNA en memoria (no relacionada con la indexación). El resto de funciones quedan en un segundo plano, de las cuales, la relevancia de los kernels esta balanceada y no se aprecia un cuello de botella en alguna función en concreto.

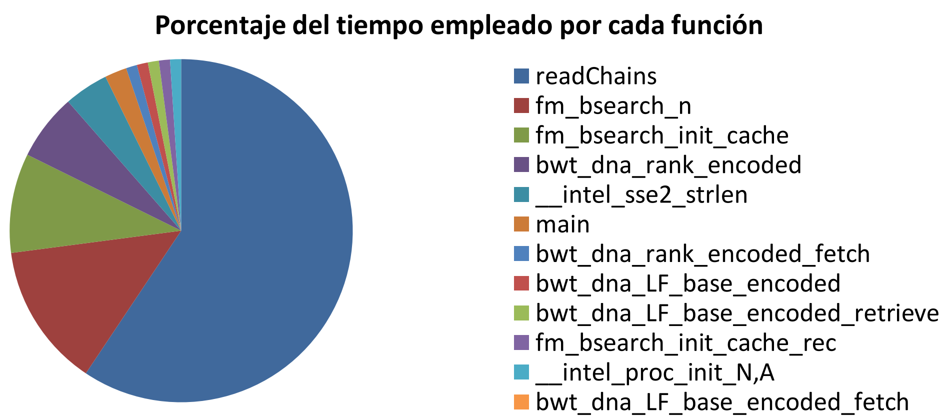

Figura 4.7 Porcentaje del tiempo empleado por cada función en la versión final

Además, en la figura 4.8 se puede apreciar como se ha reducido notablemente el número de llamadas que se realizan a los kernel más costosos y como el resto recaen sobre *bwt_dna_LF_base_encoded_fetch* que es la encargada de hacer fetch sobre una posición de memoria en el acceso entrelazado. Como conclusión podemos ver que las optimizaciones generan menos llamadas a las funciones más pesadas de la librería, ahorrando costes de ejecución.

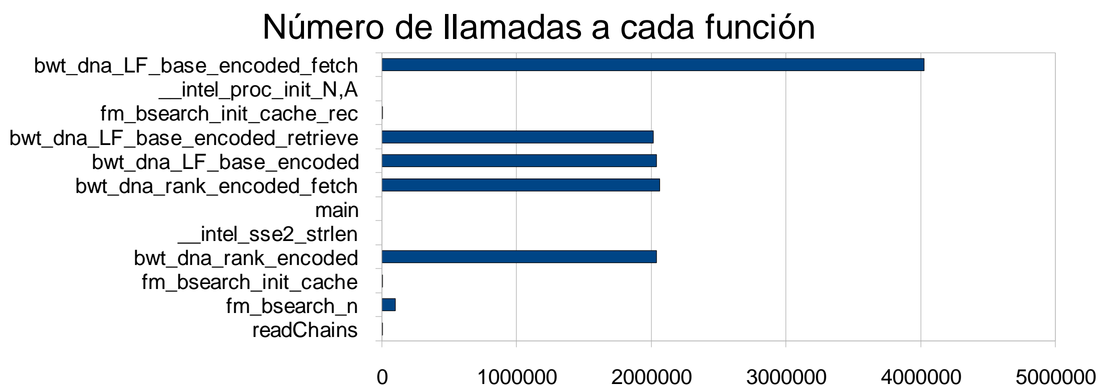

Figura 4.8 Número de llamadas a cada función en la versión final

Utilizando Vtune y comparándolo con los datos anteriormente obtenidos obtenemos las figuras 4.9 y 4.10. Podemos ver como el número de instrucciones totales que se ejecutan para realizar un acceso al índice ha descendido dramáticamente. Por ello, el número de loads que se realizan también es menor y con ello el número de referencias a la L1 cache. Ademas, debido al acceso entrelazado, podemos ver como el número de fallos en L1 y L2 cache se ha reducido drásticamente.

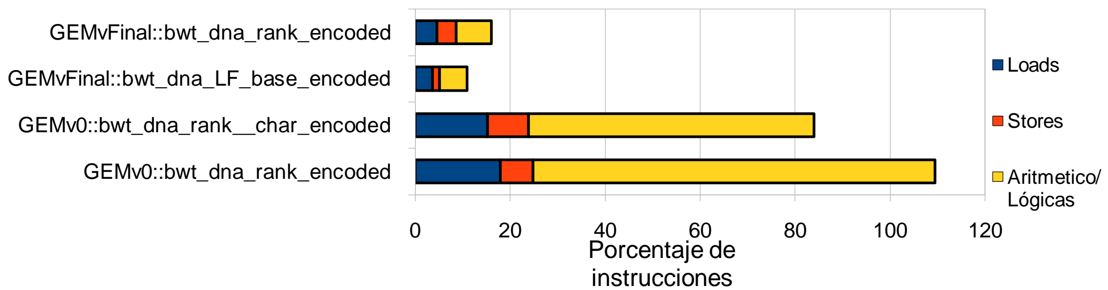

Figura 4.9 Descomposición de instrucciones de un acceso

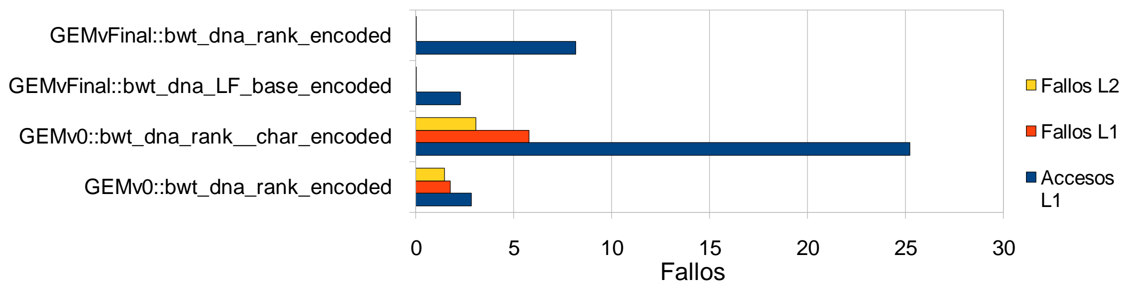

Figura 4.10 Uso de la jerarquía cache

Como era de esperar, los tiempos de ejecución de esta nueva versión se han reducido notablemente. Para una selección moderada de los parámetros de optimización, que tenga un consumo extra de memoria moderado, obtenemos la comparativa de las dos versiones en la figura 4.11. El SpeedUp para esta selección de parámetros es en media 2,5X. Es posible alcanzar Speedups de hasta 3,3X con políticas más agresivas en los parámetros de optimización.

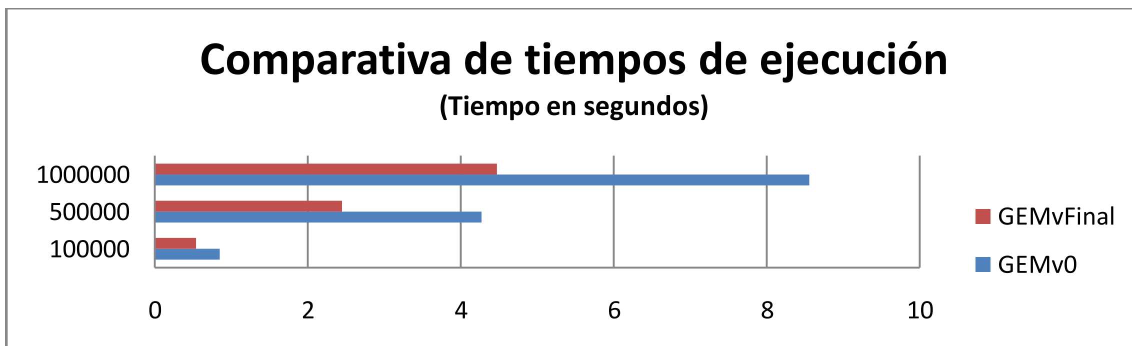

Figura 4.11 Comparativa de tiempos de ejecución

5. Análisis de la versión paralela de GEM

"'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

'Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!'

- Lewis Carroll, Through the Looking-Glass

5.1 Paralelización por división de secuencias de entrada de la versión inicial

Para la versión inicial de la librería se realizó una versión paralela. La idea fundamental de la paralelización en esta versión es el reparto de la carga de entrada entre los diferentes threads. Estos pueden atender independientemente búsquedas sin dependencias de ningún tipo. Otros planteamientos fueron descartados por ser inviables. Se planteó el paralelizar la búsqueda de cada cadena. No obstante, la carga de cada búsqueda individual es tan pequeña que no compensa los costes de comunicación con los threads. También se planteó la división del índice de referencia, pero las dependencias entre posiciones del índice lo hacen casi impracticable. La única alternativa viable no planteaba beneficios aparentes, dado que los costes de ejecución vienen determinados por la longitud de la cadena a buscar y no por el tamaño del índice.

Sobre esta versión paralela se realizaron experimentos para calcular el speedUp con diferentes políticas de asignación de datos de entrada y distinto número de threads. La figura 5.1 muestra los Speedups alcanzados utilizando reparto estático de carga, Guiado[13] y Dinámico[14] para diferentes tamaños de bloque. Podemos ver que, en general, no hay mucha diferencia dado que la granularidad de la carga es tan pequeña que el posible sesgo que pudiera causar la búsqueda de unas cadenas queda fácilmente compensado por otras del bloque de carga.

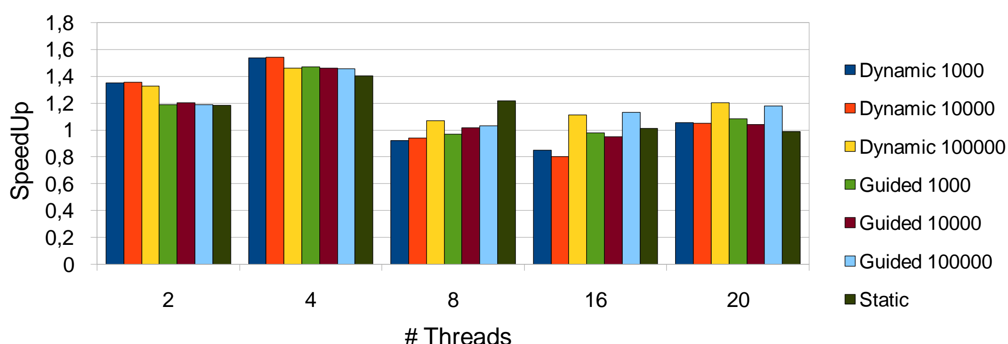

Figura 5.1 Speedup para diferentes políticas de reparto de carga

[13] Guiado. Cada thread pide un bloque de trabajos cuando está desocupado. Productor-Consumidor
[14] Dinámico. El tamaño de los bloques de carga se incrementa según se demandan los mismos.

Así, en la tabla 5.1, se muestran los speedup alcanzados según se aumenta el número de threads. Es destacable ver como el entrelazado de los hilos implementa a nivel de sistema operativo la búsqueda entrelazada de cadenas propuesta en las optimizaciones. De hecho, los speedups alcanzados están cerca de los alcanzados por la versión single-thread optimizada.

	2	4	8	16	20
OMP Auto	1,35	1,54	1,21	1,13	1,2
OMP Manual	1,86	2,87	2,89	2,89	2,8
PTHREAD	1,86	2,86	2,87	2,89	2,88

Tabla 5.1 Speedup para diferente número de threads

5.2 Paralelización de la versión optimizada

Del mismo modo, se ha paralelizado la librería para la versión optimizada de la librería. Con ello se ha podido comprobar cómo el algoritmo paralelo en memoria compartida no escala bien a partir de 2 threads (figura 5.2). Esto es debido a que el algoritmo es intensivo en memoria y su optimización es todavía más agresiva en demanda de bloques de memoria. Por ello, a partir de dos hilos el bandwith es insuficiente para atender la demanda de más hilos buscando en el índice. Debido a esto, este planteamiento paralelo no escala.

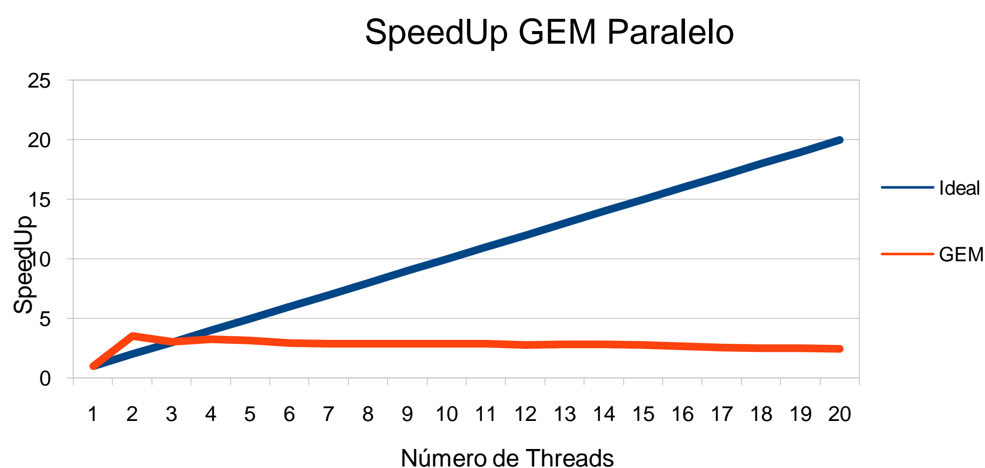

Figura 5.2 Speedup para diferente número de threads de la versión optimizada

6. Organización del trabajo

The twelve jurors were all writing very busily on slates.
"What are they doing?" Alice whispered to the Gryphon.
"They can't have anything to put down yet, before the trial's begun."
"They're putting down their names," the Gryphon whispered in reply,
"for fear they should forget them before the end of the trial."
- Lewis Carroll, Alice's Adventures in Wonderland

6.1 Metodología de trabajo

Debido al contexto en el que se enmarca este PFC y las características particulares de la librería GEM se ha seguido un proceso de retroalimentación con el autor de la misma y los centros con los se colabora (CRG y CNAG). A lo largo del periodo de trabajo se han realizado 4 reuniones para mostrar resultados y recopilar información. De este modo, muchas decisiones en las líneas de trabajo han estado condicionadas a estas "transferencias de conocimiento". En lo que a metodología de trabajo interno se refiere se ha seguido un ciclo de vida en cascada con realimentación basado en el análisis de las alternativas y propuestas y la corrección de las líneas de trabajo en base a los resultados obtenidos.

Además, ha sido necesario el establecer un entorno de trabajo para la realización de pruebas y test. En este se decidió adoptar el DNA del Homo Sapiens como índice de referencia para las pruebas (por su extensión y complejidad similar a los del entorno de producción). Por otro lado, se decidió generar los bancos de pruebas compuestos por cadenas de este mismo índice con rangos de error desde 0% hasta 100%. A su vez, estas cadenas de prueba tienen una longitud de entre 30 y 100 bases debido a que este es el rango de longitudes que las maquinas de secuenciación actuales proporcionan.

6.2 Calendario, gestión de horas y esfuerzos

Marzo	Abril	Mayo	Junio	Julio	Agosto

Color	Tarea
	Análisis del método, establecimiento de la metodología y estudio del contexto
	Análisis de la implementación y puesta a punto del entorno de trabajo
	Obtención de cifras, diagramas y mapas resultado de simulaciones y muestreo
	Propuesta y evaluación de alternativas
	Implementación y análisis de alternativas
	Implementación de la versión final
	Informe previo y presentación en el CRG de resultados
	Análisis de la estructura del índice
	Memoria

Figura 6.2 Diagrama de Gantt con el calendario

En la figura 6.2 se puede ver un diagrama de Gantt con el calendario seguido en el periodo de realización del Proyecto Fin de Carrera desde Marzo hasta finales de Agosto del 2010. Y por último, en la figura 6.3 se puede ver la distribución de horas a cada tarea. Son las tareas de implementación y obtención de cifras y simulación las que más horas han acumulado. En total han sido 913 horas dedicadas al proyecto.

Gestión de Horas y Esfuerzos

Análisis del método
Análisis de la implementación
Obtención de cifras
Propuesta de alternativas
Implementación de alternativas
Implementación de la versión final
Informe previo
Análisis de la estructura del índice
Memoria

Figura 6.3 Diagrama de horas por tarea

7. Conclusiones

"I know what you're thinking about," said Tweedledum: "but it isn't so, nohow." "Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."
- Lewis Carroll, Alice's Adventures in Wonderland

7.1 Resultados del proyecto

En primer lugar, este proyecto ha implicado tomar contacto con fundamentos y aplicaciones de la bioinformática hasta la fecha desconocidos para el proyectando. Se ha realizado un trabajo de comprensión de la naturaleza del problema de la secuenciación de DNA, sus aplicaciones e implicaciones en el ámbito de la informática.

Se han analizado las técnicas actuales de indexación de información genética y se ha profundizado en la transformada BWT y la indexación con índices FM. Además se ha estudiado la implementación de GEM de un índice FM. De este modo, se ha simulado y muestreado la ejecución de un sencillo indexador basado en GEM y se han recopilado datos sobre su ejecución. Así se ha analizado el rendimiento de la versión actual, sus cuellos de botella y secciones susceptibles de ser optimizadas.

Con los datos resultantes del análisis se han estudiado alternativas incidiendo en diversos aspectos de la librería. Desde optimizaciones de aplicación hasta optimizaciones conscientes de la arquitectura se han propuestos alternativas, se han implementado y analizados cualitativa y cuantitativamente utilizando técnicas de sampleo y simulación.

Por otro lado, se han estudiado organizaciones alternativas del índice a bajo nivel y se ha realizado un análisis del índice genómico (su estructura repetitiva, su entropía, etc) en busca de oportunidades de optimización. Se ha profundizado en la implementación de operaciones concretas de los Kernels y se han utilizado operaciones SIMD en estos (paralelismo a nivel de instrucción). Además se ha implementado y estudiado una versión de la aplicación multhreaded.

Además, como resultado de este proyecto se han implementado un conjunto de herramientas para la generación y ejecución de test, análisis del índice, reconstrucción del índice, etc.

7.2 Líneas de trabajo futuro

Resultado del trabajo realizado, el proyectando continuará el trabajo realizado trabajando para el CNAG[15] (centro con el que se colabora para este proyecto). Entre otras tareas, se tiene prevista la incorporación de un modulo de búsqueda inexacta sobre la librería actual. De este modo, se desarrollara la versión que finalmente entrará en producción y será explotada por los grupos de investigación del centro.

Además, se prevé profundizar en nuevas aproximaciones a la búsqueda exacta que introduzcan beneficios derivados de la compresión. Así como comprimir el propio índice y la búsqueda de cadenas comprimidas. Se exploraran organizaciones del índice alternativas que exploten la organización interna de las bases en los índices. Por otro lado, se exploraran alternativas paralelas no vistas en el presente proyecto

Por último, destacar que se planteará utilizar la librería para el alineamiento de secuencias como tema fundamental de la tesis.

7.3 Valoración personal

El desarrollo del presente PFC ha resultado una tarea estimulante a la par que agotadora. En este periodo he podido tomar contacto con los problemas y necesidades de la bioinformática. He profundizado conocimientos sobre estructuras de datos compactas, su análisis e implementación. Además he utilizado herramientas de simulación y muestreo que hasta la fecha no conocía. Creo que en general, todos estos conocimientos me han aportado una perspectiva más profunda sobre multitud de aéreas. Destaco sobre todo el carácter dual de este proyecto entre el área de arquitectura de computadores y el de lenguajes y sistemas. Opino que una visión global es mucho más enriquecedora e interesante. Este proyecto ha supuesto la oportunidad perfecta para poder aplicar mis conocimientos en una labor que me motiva e ilusiona.

Por último, destacar que valoro muy positivamente la oportunidad que se me ha concedido, no solo realizando este proyecto, sino ofreciéndome un puesto de trabajo por parte del Centro Nacional de Análisis Genómico. Es una oportunidad única y agradezco que se me haya ofrecido esta posibilidad de continuidad.

[15] Centro Nacional de Análisis Genómico

A. Introducción al contexto de la bioinformática

En el ámbito de la biología, la secuenciación es el proceso por el cual se obtiene la representación simbólica en A (Adenina), C(Citosina), G (Guanina) y T (Timina) de una cadena de DNA dada una muestra de material genético.

Cada una de las letras antes mencionadas codifica una base. Estas se agrupan en cadenas de DNA. Cada cadena de DNA está unida a su complementaria, formando pares de bases. De esta manera se estructuran dos secuencias de DNA (directa y complementaria). Además estas dos secuencias están orientadas (polímeros) por lo cual sus extremos pueden ser distinguidos. Estas dobles secuencias se agrupan enrolladas formando cromosomas (componentes básicos del núcleo de una célula). Por otro lado, cada tripleta de DNA forma un aminoácido. No obstante, no todas las combinaciones de tres bases forman aminoácidos existentes (válidos) y algunos de ellos son muy escasos.

En el caso del ser humano, su secuencia genética contiene 3200 millones de bases. Estas se reparten en 23 cromosomas. De toda la información genética, se cree que solo un 50% codifica información genética útil. El resto, se supone sin significado (o no conocido todavía).

En general, las mutaciones de bases o incluso secuencias cortas de bases normalmente no afectan críticamente a la célula. De hecho se cree que mucha información del DNA humano es redundante. Denominamos SNP (Single nucleotide polymorphism) a una secuencia de DNA con una alteración en uno de sus nucleótidos (base). Pese a que el DNA entre dos individuos cualesquiera se asemeja en un 99%, es ese 1% de SNPs el que nos caracteriza, diferencia, etc.

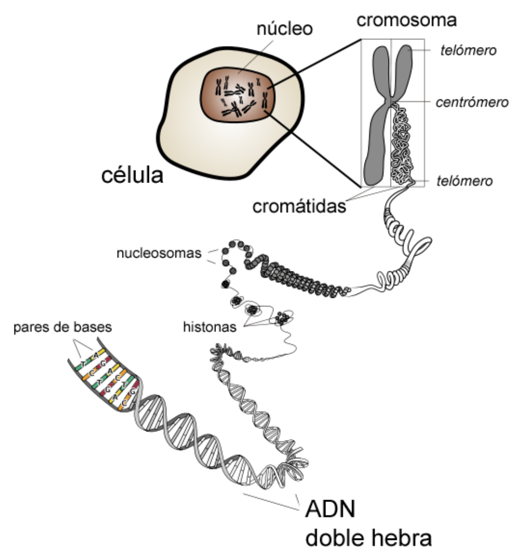

Figura A.1Estructura de la celula y el DNA

Se dispone de un patrón de referencia de DNA humano (genoma humano). Este se ha compuesto en base a la secuenciación de DNA de varios individuos, es decir, no pertenece a un ser humano en concreto. No obstante, dado el DNA de dos seres humanos no difiere en más de un 1%, sirve como base para la localización y referencia de zonas concretas del DNA, genes, etc.

Dicho lo cual, una máquina secuenciadora como AB/SOLiD o Illumina/GAII se encarga de romper esta estructura y procesarla, dando como resultado la codificación en {A,C,G,T} de pequeñas secuencias de DNA (entre 70 y 130 bases, dependiendo de la maquina). No obstante, estas maquinas secuencian dentro de un rango de error, asignando un nivel de certidumbre a cada base secuenciada. Por ello, se pueden dar vacíos entres las cadenas secuenciadas.

El problema básico que queremos tratar es el de localizar las pequeñas cadenas de bases generadas por los secuenciadores en el patrón de referencia (sea de Homo Sapiens u otro ser vivo) lo más rápido posible. El problema en sí responde al problema clásico de indexación de información. El primer objetivo es responder a las demandas generadas por las nuevas máquinas de secuenciación de DNA como AB/SOLiD o Illumina/GAII. En el siguiente gráfico se puede ver un resumen del throughput de estas máquinas en la actualidad.

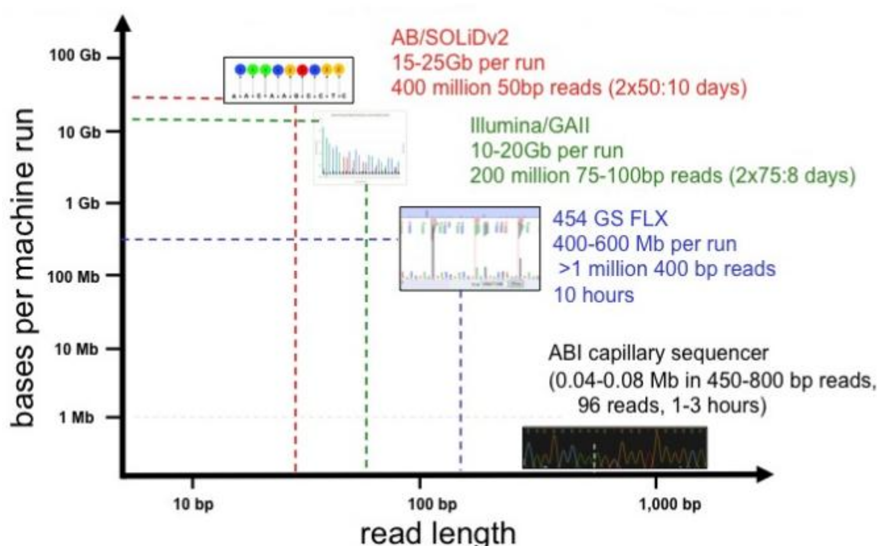

Figura A.2 Panorámica del throughput de las máquinas secuenciadoras actuales

Dependiendo de la longitud de la cadena de bases secuenciada el throughtput aumenta con cadenas pequeñas. En el peor de los casos, nos enfrentamos a un throughtput de 400 millones de secuencias de 50 bases en 10 días. Simplificando, se trata de una cota máxima de 500 cadenas por segundo. La librería GEM procesa actualmente en un procesador moderno 53.000 cadenas de entre 30 y 100 bases por segundo. En cifras absolutas, con un solo computador y la actual implementación de GEM se pueden satisfacer las necesidades de hasta 100 máquinas secuenciadoras modernas. No obstante, la maquina secuenciadora no produce información continuamente, sino que es al final de cada proceso cuando genera la información relativa a la secuenciación. Por ello, una vez generada toda la información por parte de la máquina secuenciadora, la implementación actual de la librería GEM es capaz de procesarla en 2 horas (suponiendo el procesamiento de toda la información generada).

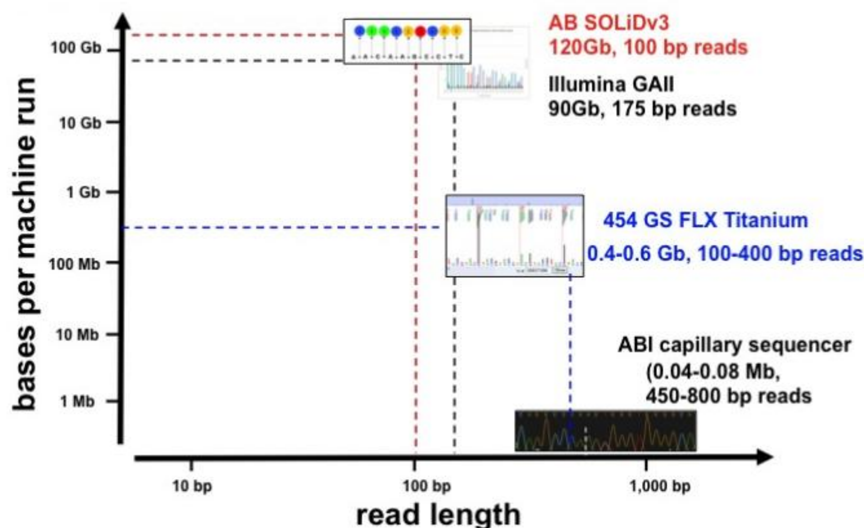

Figura A.3 Expectativas de throughput de las máquinas secuenciadoras para finales del 2010

A.1 Reunión con Paolo Ribeca (CRG. Barcelona)

En la primera reunión con Paolo Ribeca en el CRG tuvimos una primera aproximación al problema de la indexación de grandes cantidades de DNA y al contexto en el que se biológico en el que se enmarca. Paolo realizó una breve introducción a los conceptos básicos de biología necesarios para la mejor comprensión del problema computacional. Es importante destacar que el problema, pese a estar relacionado con la biología, es principalmente un problema de computación.

Paolo nos aporto básicamente unas nociones básicas de la secuenciación de DNA, sus aplicaciones y algunas cifras relacionadas (expuestas en la sección anterior). Además, nos introdujo en el ámbito del problema de la indexación de DNA desde la perspectiva de los índices FM. Además nos comentó particularidades de su implementación. Destacar que el objetivo del CRG es la secuenciación de múltiples cadenas pequeñas de DNA en su entorno de producción. Esta es una aplicación de la secuenciación que dista de la que Carmen Cons en la facultad de veterinaria aplica en sus investigaciones.

A.2 Reunión con Carmen Cons (Facultad Veterinaria. Zaragoza)

Se realizó una reunión con Carmen Cons del laboratorio de genética bioquímica (Facultad de veterinaria de Zaragoza). En esta reunión, Carmen realizó una ampliación de nuestros conceptos generales relacionados con genética.

Además, nos mostró el proceso de secuenciación que llevan a cabo en la facultad de veterinaria y como este difiere del que utilizan en el CRG. En el CRG secuencian millones de cadenas de pequeñas longitud procedentes de varias regiones del genoma. En cambio, en la facultad de veterinaria secuencian cadenas más grandes (hasta 200 bases) pero de una sección muy concreta de DNA. Es decir, secuencian unos genes específicos marcados por dos finalizadores muy concretos.

En general, pudimos ver las máquinas involucradas en el proceso y tomar contacto con las aplicaciones de la secuenciación. Entre ellas, la localización de marcadores genéticos específicos para el diagnostico de enfermedades, selección de especímenes idóneos para la cría, etc.

B. Análisis en detalle del rendimiento de la implementación

B1. Análisis del rendimiento para diferentes compiladores y opciones de compilación.

Denominaremos Sencillo Indexador con GEM (SIG) a una implementación sencilla con GEM de un buscador de cadenas en el índice de referencia (Homo Sapiens).

```
FM* a;
fm_t begin, end, position, count;
char *chains[];

for (i=0; i<=NUMBER_OF_CHAINS; i++) {
	count=fm_bsearch(a, (ch_t *)(chains[i]),
	strlen(chains[i]), &begin, &end);
	pos=fm_lookup(a,beg);
}
```

Figura B1.1 Sección principal del SIG

El SIG busca las cadenas de ACGT contenidas en chains en el indice FM a. Para ello, primero llama a la función fm_bsearch la cual devolverá la posición inicial y final en el índice FM de la cadena buscada (begin y end) y el número de ocurrencias en el índice (count). Luego, la función fm_lookup devolverá la posición de la cadena[16] en el texto original a partir de la posición de la misma en el índice FM.

Utilizando la utilidad implementada searchTest y el SIG, se han realizado una serie de test de rendimiento. La finalidad es medir el tiempo de ejecución para diferentes perfiles de compilación, compiladores y datos de entrada. Además, para cada ejecución, la utilidad searchTest generará las estadísticas de la ejecución (tiempos y métricas distinguidas) y un fichero de chequeo con los resultados de la búsqueda para verificar en toda prueba la corrección del programa frente a el impacto de las opciones de compilación (sobre todo las agresivas/peligrosas).

Todas las mediciones se han realizado en una maquina Intel Quad Core con 4GB RAM. Además, para las pruebas en las que no se especifique lo contrario, las mediciones se han realizado utilizando un juego de 1 millón cadenas de entrada de longitud normal (estándar; entre 30 y 100 caracteres) generadas aleatoriamente. El 100% de las cadenas se encuentran en el índice de referencia (Homo Sapiens) y el conjunto se ha barajado aleatoriamente. No obstante, la repetición de cadenas en el juego de pruebas en posible.

[16] Solo la primera de las ocurrencias. Bajo la premisa de que la mayoría de las pruebas se realizan con conjuntos de entrada de cadenas contenidas en el índice, el SIG no comprueba que la cadena existe en el índice (count>0).

B1.1 Rendimiento con gcc

Se han definido una serie de perfiles de compilación para GNU C Compiler (GCC). Se pretende observar el comportamiento de las diferentes opciones de compilación en los resultados del test. Los perfiles mostrados en la tabla B1.1 constituyen una muestra de referencia de todos los perfiles generados. Esta muestra es representativa para modelar el rendimiento con diferentes opciones de compilación.

Los perfiles se organizan por clase (letra) y número de perfil. Donde la letra B designa la clase de perfiles Básicos de compilación. La S se centra en el análisis del impacto de funcionalidades de compilación especificas como el inlineado, la optimización de bucles, etc. La R designa el conjunto de perfiles orientados a maximizar el rendimiento, es decir, una mezcla de las especificas (S) con las básicas (B). También se da -aunque no con gcc- la clase P, que designa las de compilación con paralelización automática. Y la clase F, que hace referencia a las compilaciones basadas en perfiladores (profiling).

En la siguiente gráfica se muestran los tiempos recogidos de la ejecución de los binarios generados con los diferentes perfiles. Se ha empleado el juego de cadenas de entrada de referencia, realizando búsquedas en un rango de 10.000 a 1 millón.

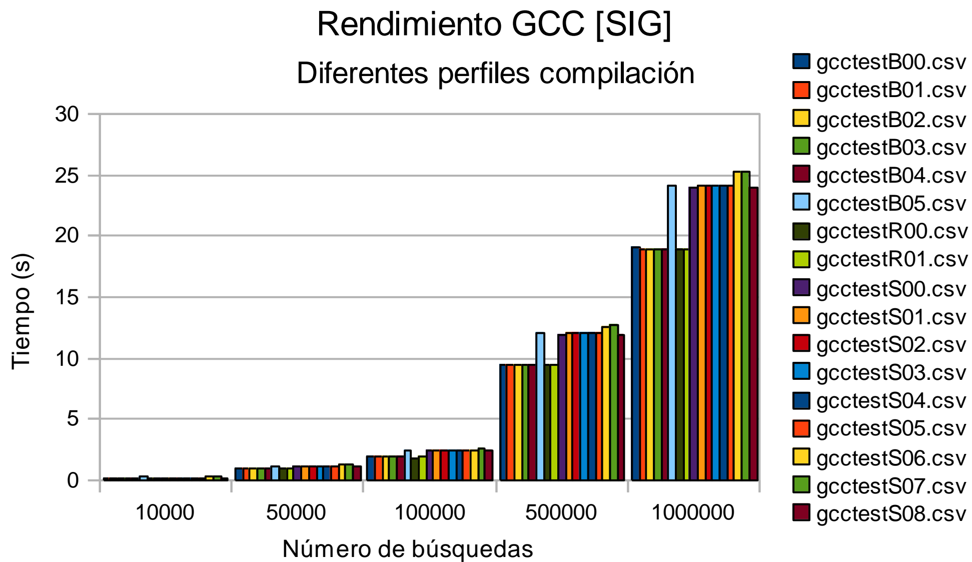

Figura B1.2 Rendimiento para GCC

Nombre	Opciones compilación / Resumen
B00	-O
	Configuración Basica O
B01	-O1
	Configuración Basica O1
B02	-O2
	Configuración Basica O2
B03	-O3
	Configuración Basica O3
B04	-O4
	Configuración Basica O4 (Paolo por defecto)
B05	-O0
	Configuración Basica Sin Optimizaciones
S00	-fomit-frame-pointer -foptimize-sibling-calls
	No guardar el frame pointer si no se necesita y optimización de llamadas recursivas
S01	-finline-functions-called-once -finline-limit= 1200
	Inlineado agresivo
S02	-fmudflapir -fdelete-null-pointer-checks
	Suprimir/Reducir la instrumentalización
S03	-fthread-jumps -fcse-follow-jumps -fcse-skip-blocks -frerun-cse-after-loop -fgcse
	Jump optimizations in branches and CSE
S04	-fgcse -fgcse-lm -fgcse-sm -fgcse-las -fgcse-after-reload
	All CSE
S05	-funsafe-loop-optimizations
	Unsafe Loop Optimizacions
S06	-fregmove -fdelayed-branch -fschedule-insns -fschedule-insns2 -fsched-spec-load -fsched-stalled-insns -fsched-stalled-insns-dep
	Reorganización de registros y de instrucciones
S07	-fregmove -fdelayed-branch -fschedule-insns -fschedule-insns2 -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns -fsched-stalled-insns-dep -fsched2-use-superblocks -fsched2-use-traces
	Reorganización de registros y de instrucciones
S08	-finline-functions-called-once -finline-limit= 1200 -fomit-frame-pointer -fforce-addr
	Inlineado agresivo y uso intensivo de registros
R00	-O3 -fgcse -fgcse-lm -fgcse-sm -fgcse-las -fgcse-after-reload -fmudflapir -fexpensive-optimizations
	Opciones de optimización que por defecto no se incluyen en O3
R01	-O3 -fgcse -fgcse-lm -fgcse-sm -fgcse-las -fgcse-after-reload -fmudflapir -fexpensive-optimizations -funsafe-loop-optimizations
	Opciones de optimización No seguras + O3

Tabla B1.1 Diferentes perfiles de compilación con gcc

Se puede apreciar como destacan dos clases de rendimiento bien diferenciadas. Dentro de la primera -los peores rendimientos- están los asociados a los perfiles con optimizaciones especificas. Dado que estos perfiles atacan características concretas de la compilación es comprensible que muestren un desempeño peor. No obstante, en la figura B1.3, se puede observar como todos los perfiles de optimizaciones especificas (menos el S06 y S07) obtienen tiempos en torno a los 24s, el mismo tiempo que se obtiene sin optimizaciones (B05). Por ello, se puede decir que ninguna de las optimizaciones especificas mejora el rendimiento. Incluso la S06 y S07 (reorganización de código e instrucciones) empeoran el desempeño (un 4% más lento que B05).

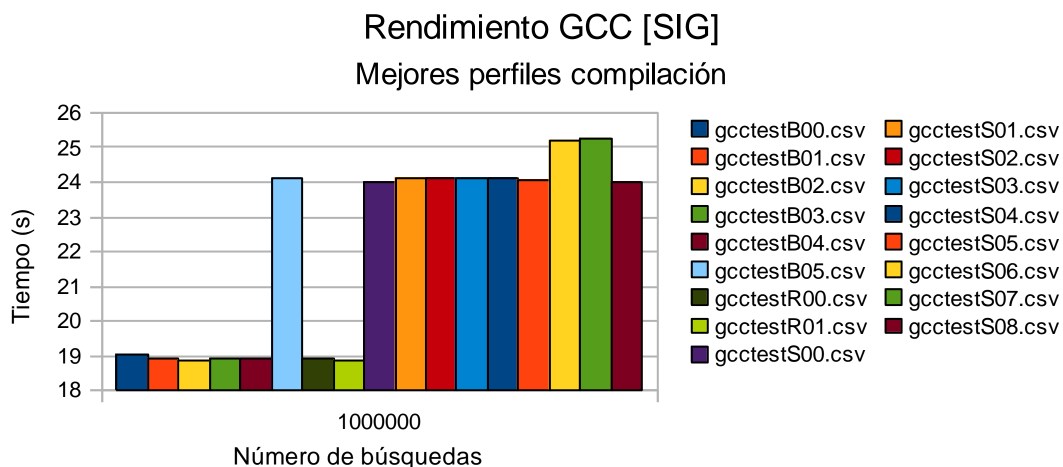

Figura B1.3. Detalle mejores perfiles de compilación con GCC

Por otro lado, en el segundo grupo, tenemos los perfiles que consiguen mejorar el rendimiento[17]. Estos obtienen todos un rendimiento en torno a los 19s (22% aceleración con respecto a B05). No obstante, la diferencia en los tiempos de ejecución no es lo suficientemente relevante como para ser considerada o atribuida a alguna opción especifica de compilación sobre las demás del grupo. Las diferencias pueden ser atribuidas perfectamente a la carga del sistema.

El hecho de que los tiempos de ejecución se agruparan tan claramente en dos grupos nos llevó a inquirir en la necesidad de analizar el código ensamblador generado por los perfiles. En este punto pudimos comprobar como muchos perfiles generaban el mismo código. O lo que es lo mismo, muchas opciones de compilación no aplican o no consiguen ninguna modificación.

[17] Se supone el perfil B05 como referencia (sin optimizaciones -O0)

El hecho de que no solo no se obtengan diferentes tiempos de ejecución, sino que tampoco se obtengan diferentes códigos revela la poca capacidad de actuación del compilador con el código. Esto pone de manifiesto que los posibles planteamientos de optimización del código no deben focalizarse en transformaciones generales del código[18]. En general, podemos establecer que con una compilación optimizada con gcc se puede obtener un throughput de hasta **53.000 cadenas por segundo**.

B1.2 Rendimiento con icc

Del mismo modo que en el apartado anterior, se han definido una serie de perfiles de compilación para Intel C Compiler (ICC). Se pretende observar el comportamiento de las diferentes opciones de compilación en los resultados del test utilizando el compilador propietario de Intel. El planteamiento que sustenta el uso de este compilador está basado en que icc es el compilador especifico de la arquitectura Intel. Por ello y por varios casos documentados de éxito, se considera que el compilador es candidato a generar mejores tiempos que gcc.

Destacar que para que compilara con icc se han eliminado funciones de la librería que no compilaban con este compilador al hacer uso de extensiones de GCC como las *nested functions*. En concreto, estas extensiones se encuentran en la función de creación de índices. No obstante, al estar fuera del propósito del estudio previo, no se ha tenido en cuenta y simplemente se ha prescindido de ella. Pese a todo, se han planteado alternativas de implementación para conseguir que todo el código sea compatible con icc.

A continuación, tabla B1.2, se muestra un subconjunto de referencia de los perfiles generados para icc. Esta muestra es representativa para modelar el rendimiento con diferentes opciones de compilación. Las convenciones de nombrado son las mismas que en los apartados anteriores.

[18] Basadas en manipulaciones de arboles sintácticos como hace el compilador

Nombre	Opciones compilación / Resumen
B00	-O0 Configuracion Basica O0
B01	-O1 Configuracion Basica O1
B02	-O2 Configuracion Basica O2
B03	-O3 Configuracion Basica O3
B04	-fast Configuracion Basica Fast de Intel
S00	-fast -unroll -unroll-aggressive -funroll-loops Desplegado agresivo de bucles
S01	-fast -scalar-rep -complex-limited-range Scalar Replacement
S02	-fast -ansi-alias -alias-const -fargument-alias Aliasing Optimizations
S03	-fast -ansi-alias -alias-const -fargument-alias Aliasing Aggressive Optimizations
R00	-fast -mtune=pentium4 -march=pentium4 -mssse3 -funroll-loops -scalar-rep -ansi-alias -complex-limited-range -opt-multi-version-aggressive -vec-guard-write -opt-subscript-in-range -opt-prefetch=4 -inline-level=2 -align -w

Tabla B1.2 Diferentes perfiles de compilación con icc

A continuación se muestran los resultados obtenidos con los perfiles especificados. Se aprecia que todos los perfiles generan binarios con similar tiempo de ejecución, mejorando un 25% el tiempo sin optimizaciones (iccB00). No obstante, a diferencia de gcc, no se dan tantos casos de código ensamblador idénticos generado para diferentes perfiles. Pese a seguir existiendo casos donde opciones diferentes de compilación generan los mismos ejecutables. Ya sea bien porque no aplica o porque la modificación no es sustancial.

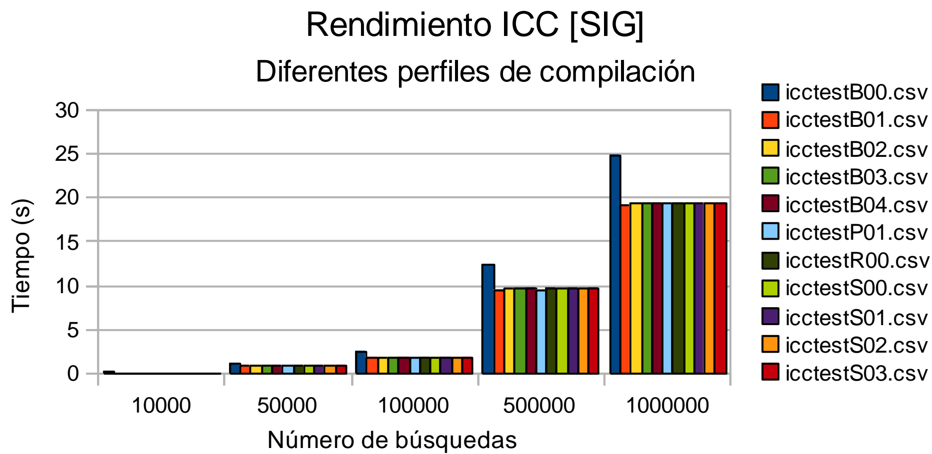

Figura B1.4 Rendimiento Icc

De forma más detallada, podemos ver en la figura B1.5 el rendimiento de los perfiles para icc con búsquedas de 1 millón de cadenas. Es importante tener en cuenta el rango de tiempo del gráfico. Ver que no varía en más de 0,4 segundos. Por lo cual, no es más significativo más allá de las variaciones de carga en el sistema (Scheduling).

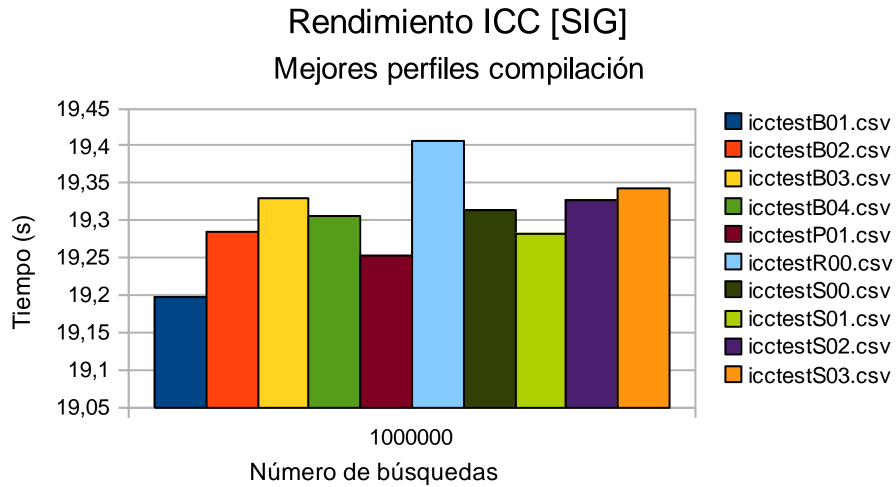

Figura B1.5. Detalle mejores perfiles de compilación con ICC

En este caso, podemos establecer que con una compilación optimizada con icc se puede obtener un throughput de hasta **52.000 cadenas por segundo**.

B1.2 Comparativa gcc/icc

Una vez obtenidos los resultados para los diferentes perfiles de compilación de ambos compiladores, se presenta una comparativa. En ella, solo se han incluido las configuraciones más eficientes de las anteriores expuestas (una muestra representativa).

Figura B1.6 Comparativa de rendimiento al compilar con icc y gcc

En general, se aprecia una marcada -aunque no relevante- eficiencia en los binarios compilados con gcc (entre un 3% y 1% más eficiente). Según el cuadro 4.5, esta diferencia tiene a desaparecer con volúmenes grandes de datos. Por todo ello, no se considera relevante, pese a ser una sorpresa que icc no obtuviera mejor desempeño que gcc.

LOAD	10000	50000	100000	500000	1000000
MAX Tex gcc	0,2	0,95	1,9	9,51	19,02
MIN Tex gcc	0,19	0,94	1,88	9,42	18,86

Tabla B1.3 Tabla con las diferencias porcentuales relativas entre icc y gcc

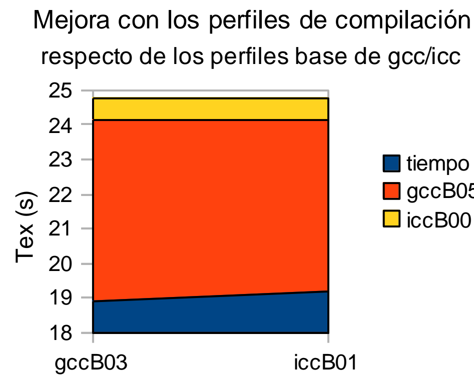

Figura 4.7 Mejora de rendimiento con los perfiles de compilación

B2. Comportamiento para diferentes configuraciones de cadenas de búsqueda

Por otro lado, se ha querido comprobar como afecta al rendimiento las propiedades de las cadenas de entrada. Para ello, se han elaborado diferentes juegos de cadenas de entrada variando su longitud y el número de ellas contenidas en el índice de referencia.

En la primera figura podemos ver como la variación de hits[19] afecta a los tiempo de ejecución realizando búsquedas de entre 300.000 y 1 millón de cadenas. La longitud de las cadenas de entrada de este juego oscila entre las 30 y 100 bases[20] (longitud estándar usada en el presente estudio).

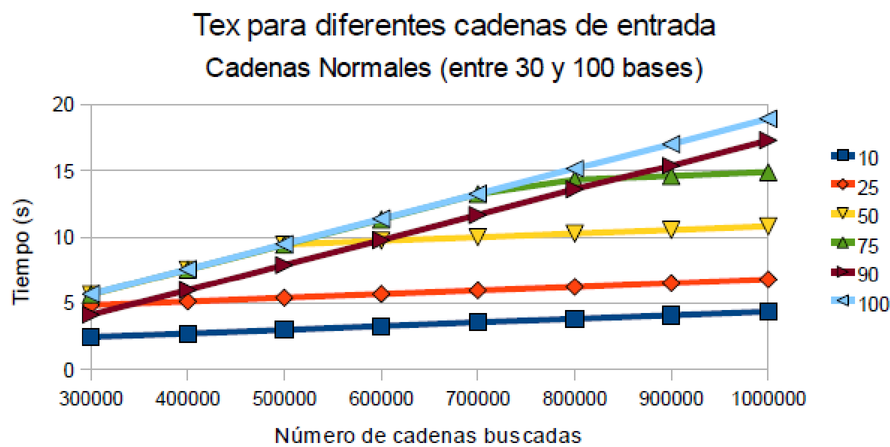

Figura B2.1 Rendimiento de SIG para diferentes configuraciones de entrada de entre 10% y 100% hits

[19] El termino “Hits” hace referencia al porcentaje de cadenas de la entrada que están contenidas en el índice de referencia

[20] Cada una de las letras {A,C,G,T} que componen las cadenas de entrada

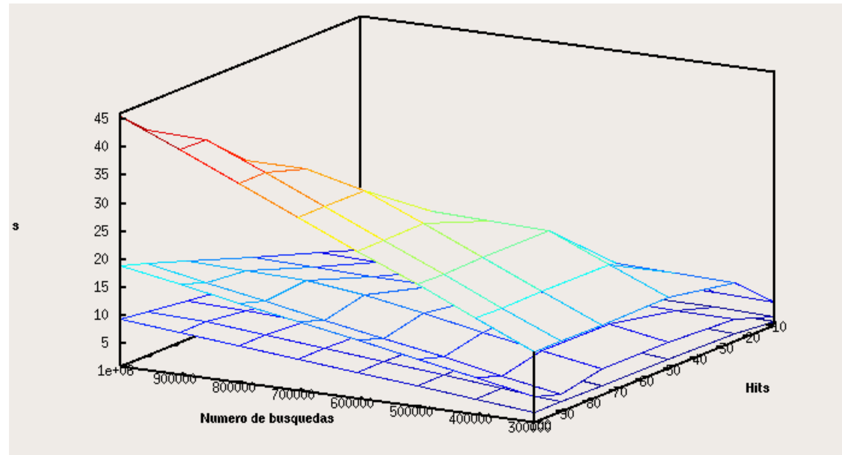

Figura B2.2 Rendimiento de SIG para diferentes configuraciones de entrada para cadenas pequeñas, medianas y grandes

En las gráficas anteriores se puede apreciar la tendencia esperada; un incremento de coste según el aumento del porcentaje de hits. Este incremento no es perceptible claramente hasta un conjunto de búsquedas de entre medio y un millón de cadenas de entrada. No obstante, no es un crecimiento inesperado. Dado que simplemente refleja el costo añadido de buscar una cadena que sí se halla en el índice. Debido al propio algoritmo de búsqueda, una cadena contenida en el índice necesitará más iteraciones en el bucle principal de fm_bsearch. Por contra, una cadena no contenida solo necesitará un número de iteraciones suficiente para descartarlo como sufijo de cualquier cadena del índice (menor al número de caracteres la cadena buscada).

A su vez, en la figura B2.2, se pueden ver los tres planos que modelan el rendimiento para cadenas de diferente longitud. La superior mide el tiempo de ejecución con cadenas de longitud grande (100-200 bases), la del medio con cadenas medianas (30-100 bases) y la de más abajo cadenas pequeñas (5-30 bases). Se apreciar como los crecimientos de los tres planos varía según se aumenta el número de hits, el número de búsquedas y la longitud de la cadena buscada. A su vez, este también es un resultado esperado. Dado que cuanto más largas sean las cadenas, más búsquedas realicemos y mayor número de ellas estén contenidas en el índice mayor coste conllevará para el algoritmo.

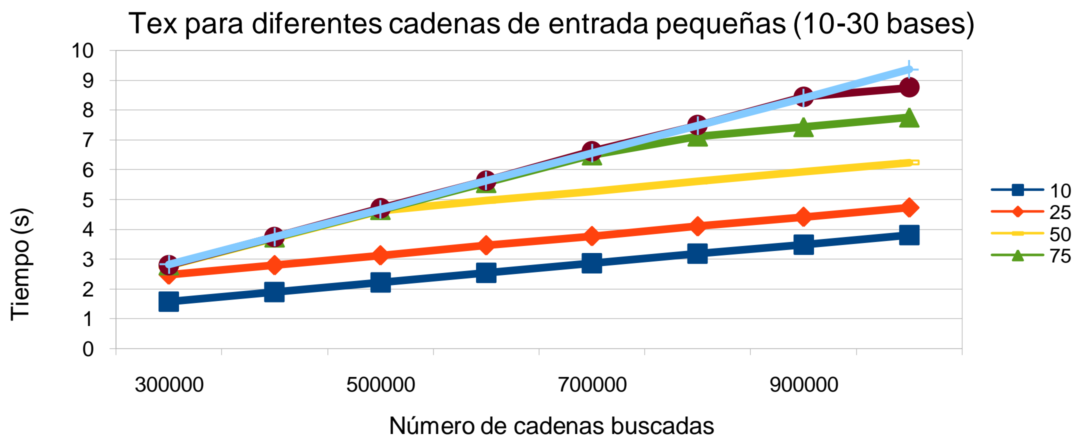

Figura B2.3 Tex para diferentes cadenas de entrada pequeñas

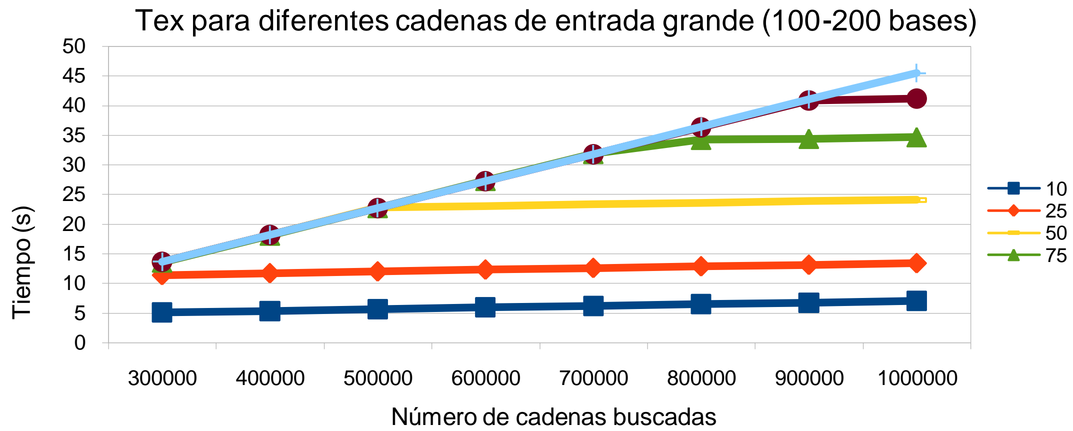

Figura B2.4 Tex para diferentes cadenas de entrada grandes

Las figuras anteriores vienen a destacar como la diferencia del tiempo de ejecución entre búsquedas de mayor o menor número de hits solo se manifiesta ante conjuntos lo suficientemente grandes. Por ejemplo, la serie de 50% hits y la de 100% hits se distancian desde conjuntos de más de 500.000 cadenas de entrada. Mientras que las de 90% con la de 100% solo lo hace a partir de 900.000.

No obstante, en las pruebas realizadas con volúmenes grandes de datos se ha podido comprobar como esta diferencia entre series con distinto porcentaje de hits es constante y proporcional porcentaje de hits.

B3. Análisis de los cuellos de botella

B3.1 Rendimiento de fm_bsearch

De forma más detallada, se ha analizado el impacto en el rendimiento total del desempeño de la función fm_bsearch. Recordar que la función fm_bsearch era la encargada de buscar en el índice FM la cadena. Esta función acota el intervalo del índice donde se encuentran los sufijos que contienen como prefijo la cadena buscada.

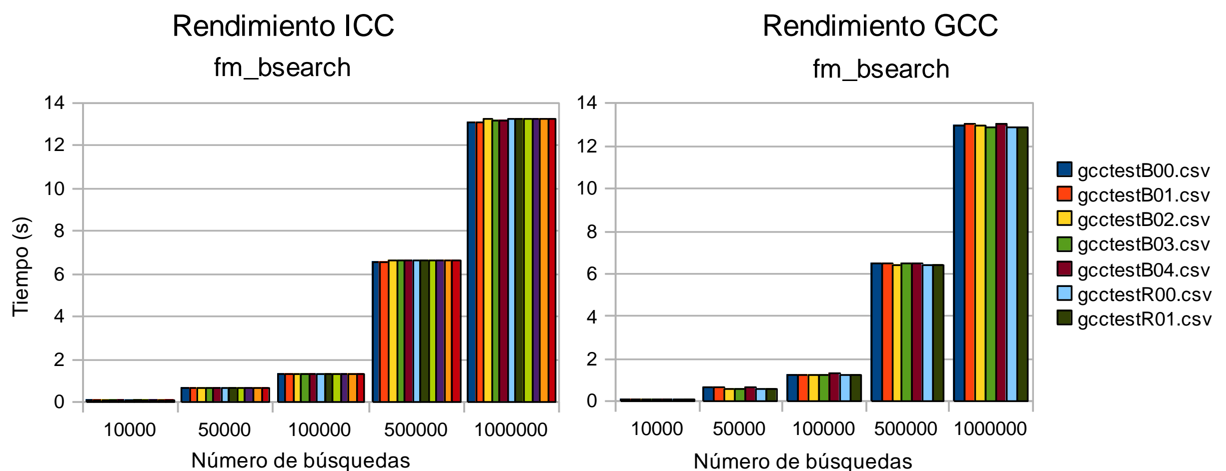

Figura B3.1 Rendimiento de fm_bsearch con los diferentes perfiles de compilación y compiladores

Como en el caso del SIG, el crecimiento en el coste es lineal y la función se ve poco afectada por los perfiles de compilación.

B3.2 Rendimiento de fm_lookup

A su vez, también se ha analizado en detalle el rendimiento de la función fm_lookup. Esta función es la encargada de traducir las posiciones en índice comprimido devueltas por fm_bsearch y las posiciones reales en el índice sin comprimir.

La relevancia de esta función está fuertemente supeditada a la forma del programa que la utilice. Esto puede parecer una obviedad. No obstante, hay que destacar que el SIG solo "decodifica"[21] la primera posición del intervalo devuelto por fm_bsearch (por definición). En otros escenarios (como en el caso de un SIG-Completo), el objetivo puede ser el de recuperar todas las posiciones del texto original donde se encuentra la cadena buscada. En estas circunstancias, la función fm_bsearch pierde relevancia en tanto y cuanto que el rendimiento global queda supeditado al de fm_lookup (que se ejecuta tantas veces como ocurrencias de la cadena devuelve fm_bsearch).

Como en los casos anteriores, el crecimiento en el coste es lineal y la función se ve poco afectada por los perfiles de compilación.

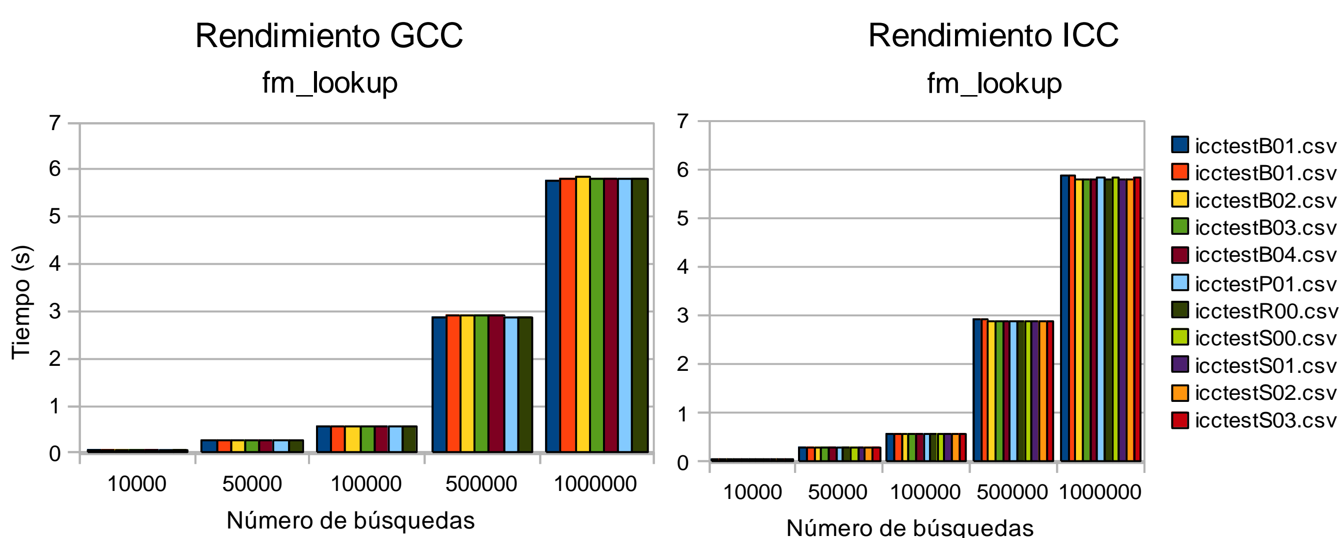

Figura B3.2 Rendimiento de fm_bsearch con los diferentes perfiles de compilación y compiladores

B3.3 Informe del Profiler gprof

Con el propósito de analizar en que partes del código invertimos más tiempo a fin de identificar cuellos de botella, se han realizado varias ejecuciones con el código instrumentalizado utilizando gprof. Recordar que estos resultados se realizaron en una maquina Intel Quad y solo han de ser tomados entre ellos como referencia. De la ejecución del SIG con el profiler se han obtenido los siguientes resultados.

[21] La transformación de índices FM a índices del texto original es realizada por fm_lookup

module	name	% time	cumulative seconds	self seconds	calls
BWT-DNA.c	*bwt_dna_rank_encoded*	67,34	143,03	143,03	1289822240
	bwt_dna_rank__char_encoded	25,26	196,69	53,66	274894150
	bwt_dna_LF_base_encoded	2,82	202,69	6	1299822240
FM.c	*fm_lookup*	2,61	208,22	5,54	10000000
	fm_bsearch	1,16	210,69	2,47	10000000
BWT-DNA.c	*bwt_dna_LF*	0,28	211,28	0,59	274894150
	bwt_dna_delete	0,19	211,69	0,41	
	bwt_dna_rank__char	0,18	212,07	0,39	
btest.c	*main*	0,09	212,26	0,19	
BWT-DNA.c	*bwt_dna_char*	0,06	212,39	0,13	
	bwt_dna_LF_base	0,05	212,5	0,11	
btest.c	*readChains*	0	212,51	0,01	1
FM.c	*fm_lookup_stats*	0	212,51	0,01	
typedef.h	*cblog*	0	212,51	0	2
MM.c	*mm_read*	0	212,51	0	2
BWT-DNA.c	*bwt_dna_read*	0	212,51	0	1
FM.c	*fm_read*	0	212,51	0	1

Tabla B3.1 Resultados de la ejecución de SIG con gprof

Simplificando el cuadro anterior y dejando solo la información relevante, podemos ver el en la tabla B3.2 los resultados resumidos.

module	name	% time	cumulative seconds	self seconds	calls	self us/call	total us/call
BWT-DNA.c	*bwt_dna_rank_encoded*	67,34	143,03	143,03	1289822240	0,1109	0,1648
	bwt_dna_rank__char_encoded	25,26	196,69	53,66	274894150	0,1952	0,7731
FM.c	*fm_lookup*	2,61	208,22	5,54	10000000	0,5540	21,2510
	fm_bsearch	1,16	210,69	2,47	10000000	0,2470	21,2510

Tabla B3.2 Resumen resultados de la ejecución de SIG con gprof

En la tabla B3.2 podemos ver como el mayor porcentaje del tiempo recae sobre *bwt_dna_rank_encode*. Esta función es llamada por *fm_bsearch* y se encarga de calcular el "*rank*" de un carácter en el índice[22]. La función *bwt_dna_rank_encode* es un componente fundamental del algoritmo de indexación basado en índices FM y el cuello de botella en el SIG. A su vez, también es la función sobre la que se realizan más llamadas. Y por contra, estas cuestan menos tiempo que las llamadas a *bwt_dna_rank__char_encoded*, llamada desde fm_lookup.

[22] Rank de c hasta i es el número de ocurrencias de c en BWT[0..i]. También figura en la literatura como función occ(BWT,c,i).

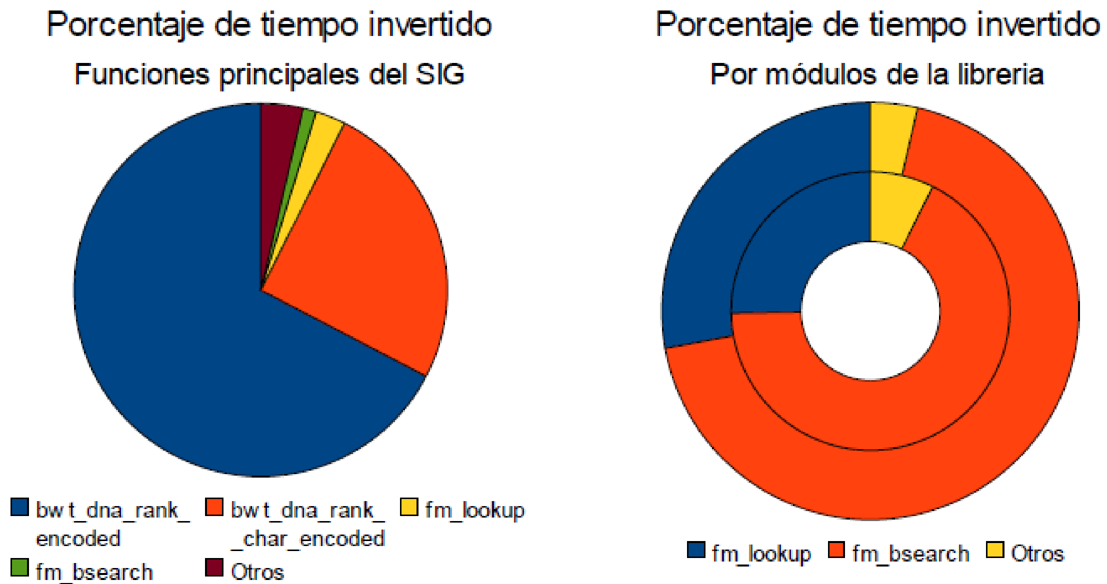

Figura B3.3 Porcentaje del tiempo invertido en cada función del SIG

Es importante destacar una vez más el coste tan pequeño de recuperar una sola cadena. Puesto que merece tener en cuenta este aspecto y sus repercusiones más adelante en la implementación de una versión paralela.

Por otro lado, reiterar la advertencia de que el SIG solo recupera la posición en el índice original de la primera ocurrencia encontrada. En el caso de un SIG-completo tendríamos otro escenario, mostrado en la siguiente figura. Podemos ver como la función bwt_dna_rank__char_encoded toma relevancia y por ende fm_lookup se convierte en el cuello de botella. Insistir en que es fundamental centrar el caso de aplicación de estas funciones. Las cifras están fuertemente ligadas a los escenarios de uso y el rendimiento depende fuertemente de estos factores.

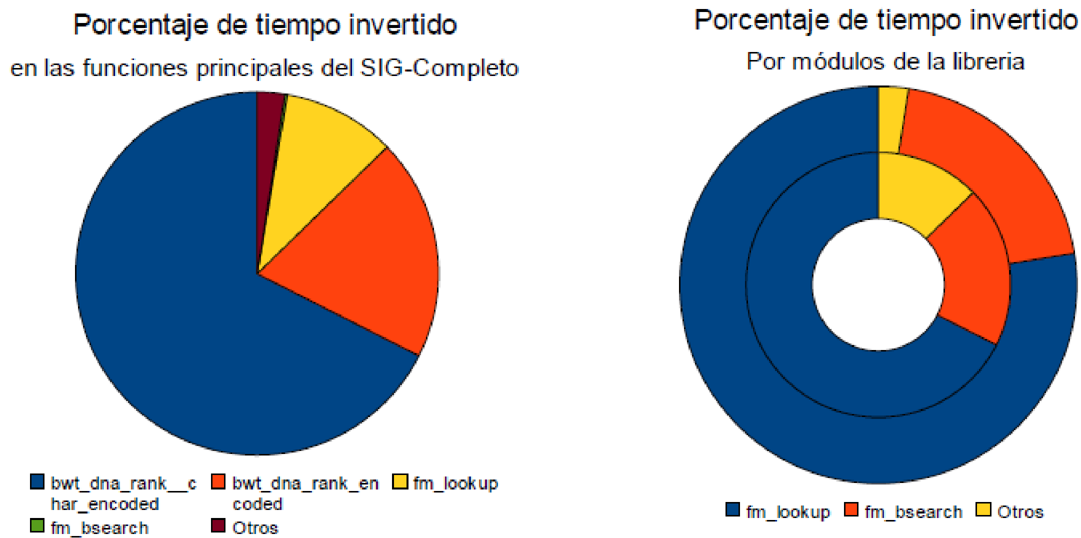

Figura B3.4 Porcentaje del tiempo invertido en cada función del SIG-Completo

C. Análisis en detalle del comportamiento en memoria

Este apartado tiene como objetivo exponer los resultados obtenidos del análisis del uso de la memoria. Para ello, se han realizado diversas simulaciones con el fin de determinar magnitudes cuantitativas (como los fallos/aciertos en cache) y cualitativas (disposición de los accesos al índice).

C1. Simulaciones para el dimensionado de la cache

En esta sección se exponen los resultados extraídos de la simulación con cacheGrind del SIG, fm_bsearch y fm_lookup para diferentes tamaños de cache. Se ha realizado un análisis de los accesos a memoria para diversos tamaños de L1 y L2. Además, se ha realizado un análisis más detallado para las configuraciones hardware de referencia.

Se ha utilizado btest para realizar las simulaciones. Este programa es una versión mínima del bucle de prueba de búsqueda en el índice FM (fm_bsearch + fm_lookup). Se ha extraído todas las funcionalidades de medición de tiempos y extracción de estadísticas para no alterar los resultados de la simulación. El objetivo es el de quitar al máximo la carga de instrumentalización.

Utilizamos cacheGrind para simular dos niveles de cache L1 y L2 con diferentes tamaños de L2. El nivel L1 con tamaños de 64K y 128K (tamaño total de la cache separada).

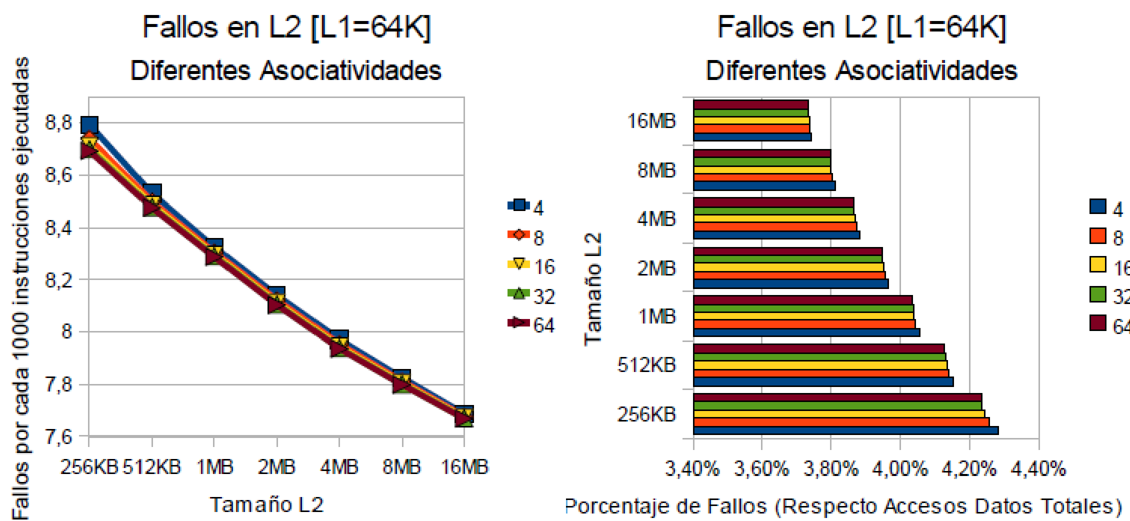

Figura C1.1 Fallos totales de datos en L2 y porcentuales al número de accesos para L1 64KB [Fallos en L1 = 200.000.000]

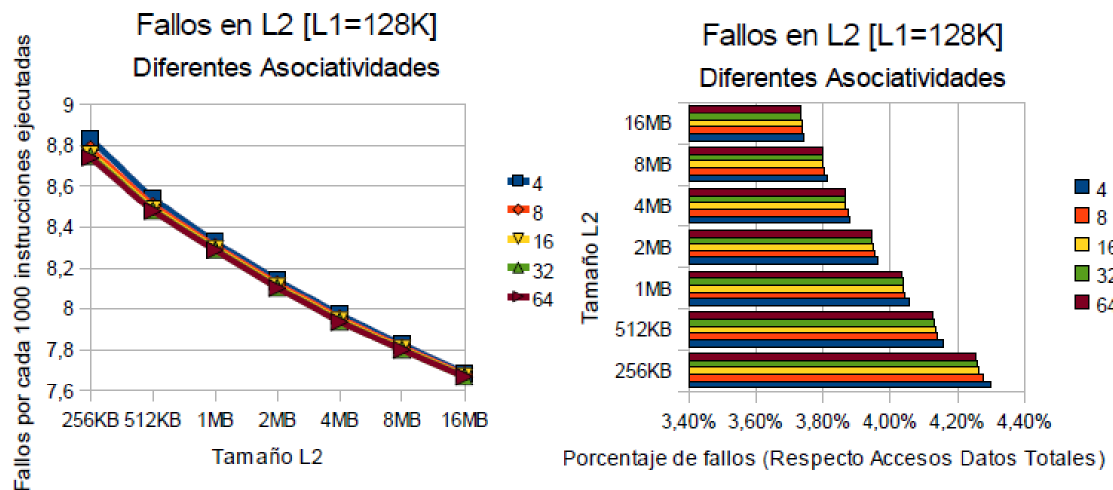

Figura C1.2 Fallos totales de datos en L2 y porcentuales al número de accesos para L1 128KB [Fallos en L1 = 188.000.000]

Como se puede ver, el resultado era esperado, al aumentar el tamaño de L2 se reducen los fallos. Así pues, al aumentar la asociatividad se producen menos reemplazos y por ende menos fallos. Los resultado se centran en la tasa de fallos en acceso a datos, dado que el número de fallos en L1 y L2 de instrucciones son solo 1062 y 1060 respectivamente (obligatorios).

También podemos ver los resultados de la simulación con un nivel de cache L1 variando la asociatividad.

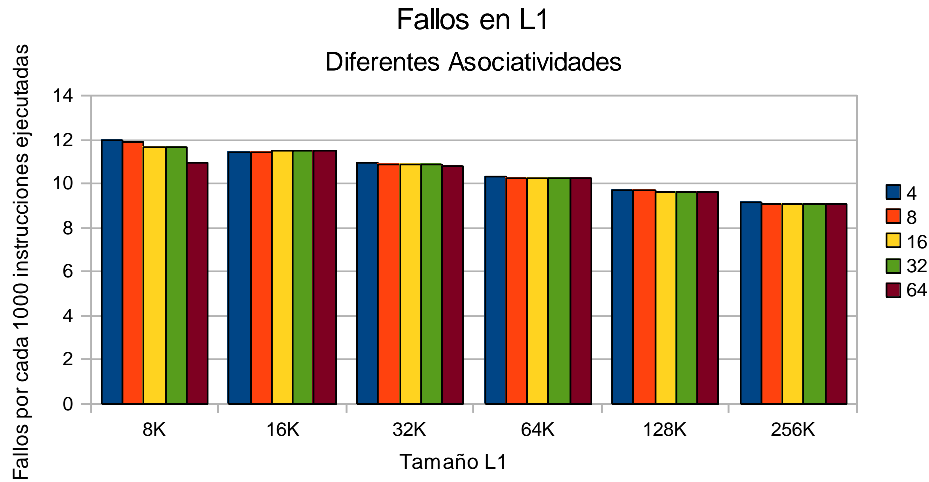

Figura C1.3 Fallos totales en L1

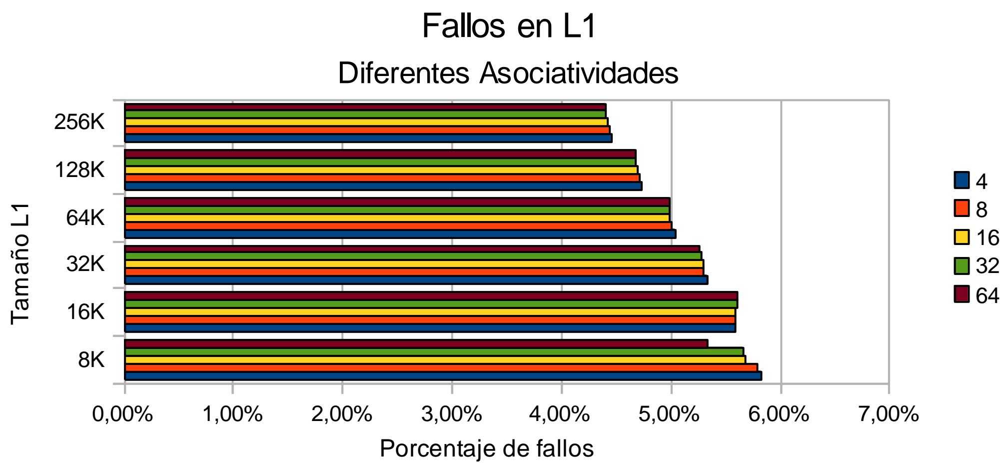

Figura C1.4 Porcentaje de fallos en L1

Del mismo modo que antes, tampoco sorprende que el número de fallos en L1 descienda según aumenta el tamaño de la cache o la asociatividad. Por último, podemos ver los resultados de variar el tamaño de bloque utilizado.

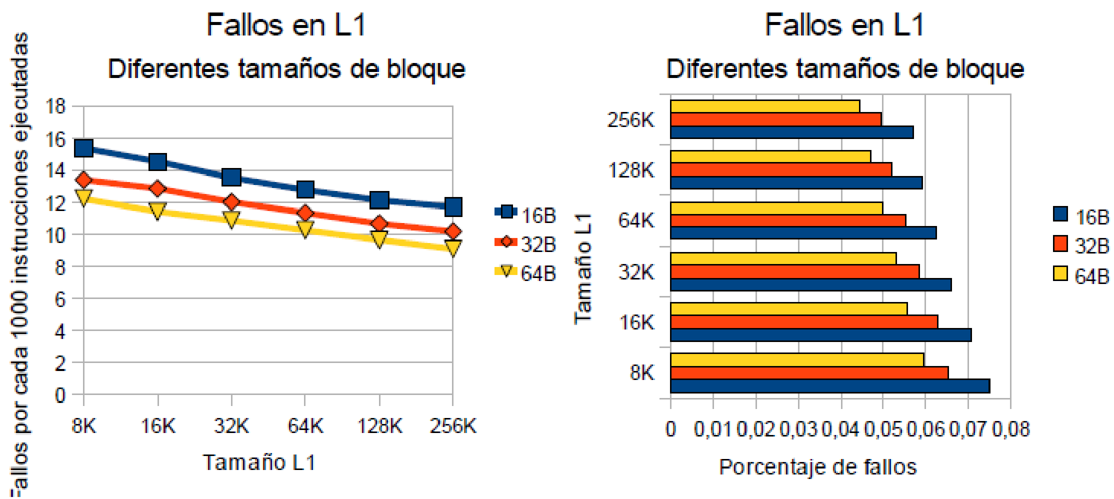

Figura C1.5 Fallos en L1 para diferentes tamaños de bloque

Podemos ver como, en general, la tasa de fallos se reduce con el aumento del tamaño del bloque. A modo de conclusión, podemos afirmar que el rendimiento de la librería no está limitado por el tamaño, asociatividad o bloque de la cache. Su dimensionado no afecta de forma crítica el rendimiento. Ver que el dimensionado no es la causa subyacente de los fallos en L1 y L2, pese a ser una librería intensiva en memoria.

Por último, podemos observar en el gráfico de saltos con los datos que nos proporciona la simulación. Ver que la mayoría de los saltos son predichos, siendo una minoría los no acertados. Esto pone de manifiesto que los condicionales instalados en el camino crítico no deberían ser foco de atención de cara a la implementación de optimizaciones.

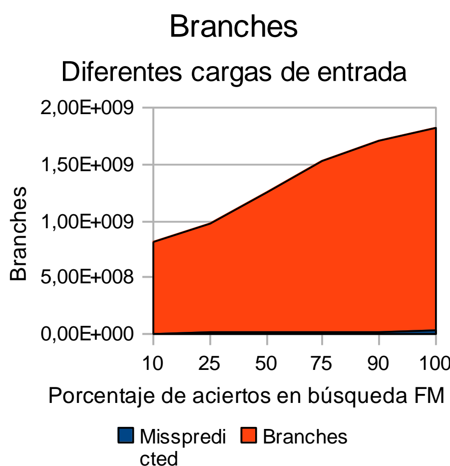

Figura C1.6 Aciertos y Fallos en predicciones de saltos

C2.Análisis de los accesos al índice y distribución de las cadenas en el mismo

Esta sección tiene como propósito realizar un análisis de la distribución de las cadenas en el índice, su acceso en las búsquedas y – en general – cualquier información relacionada con la estructura y uso del índice (tanto de forma cuantitativa como cualitativa).

C2.1 Gráficos de próximos accesos para la función fm_bsearch

La función fm_bsearch, como ya se ha comentado, se encarga de realizar la búsqueda de la posición en el índice FM de la cadena (Backward-Search). El esquema algorítmico asociado a fm_bsearch se muestra a continuación.

```
entero32 fm_bsearch(FM indiceFM,cadena clave, entero32 longitudCadena,
                    entero32 principio, entero32 fin) {
  entero32 lo=0, hi=indiceFM.longitud, i=longitudCadena;
  entero16 caracterBusqueda;
  i--;
  while ( i > 0 ) {
    caracterBusqueda = codificar(clave[longitudCadena]);
    if (noEstaEnAlfabeto(caracterBusqueda) || hi==lo) {
      principio=0; fin=0; return 0;
    }

    lo=_BWT_LF_base_encoded(indiceFM.indiceBWT,caracterBusqueda,lo);
    hi=_BWT_LF_base_encoded(indiceFM.indiceBWT,caracterBusqueda,hi);

    i--;
  }
  principio=lo;
  fin=hi;
  return hi-lo;
}
```

Figura C2.1 Esquema algorítmico de fm_bsearch

Como se puede apreciar, la función consta de un bucle principal con dos llamadas a "_BWT_LF_base_encoded" totalmente independientes una de la otra. Su propósito – como ya se ha expuesto – es de acotar el rango de cadenas en el índice que contienen el prefijo clave[i..longitud(clave)][23]. De esta forma, llamando a "_BWT_LF_base_encoded"[24] consiguen la siguiente posición del índice a buscar en el proceso de acotación.

[23] Por lo tanto se cumple el invariante lo<=hi
[24] Función encargada de extraer la información del índice FM (comprimido)

Es dentro de la función "_BWT_LF_base_encoded" donde se produce ese 4% de fallos en L1/L2 expuesto anteriormente. Y es sin duda debido a los accesos demandados desde esta función en la capa de FM. Debido a esto se consideró necesario estudiar los accesos producidos al índice; su ordenación, su frecuencia, etc.

En primer lugar, se expone el gráfico de próximos accesos. Este gráfico tiene como finalidad mostrar los próximos accesos que se realizaran desde una posición dada del índice. Es decir, la función "_BWT_LF_base_encoded" toma el índice FM, el siguiente carácter (A,C,G o T) y la posición anterior explorada del índice (posición inicial). Entonces, fijada una posición inicial del indice (desde 0 hasta 2.858.045.942) podemos acceder con una iteración a 4 posiciones del índice. Con dos iteraciones podremos acceder a 16 posiciones y así en adelante.

Así, podemos representar el primer gráfico de próximos accesos. Este muestra en las ordenadas las posiciones del índice iniciales (agrupadas en bloques de una cierta resolución[25]) y en las abscisas las posiciones del índice destino. Si realizamos la búsqueda de la carga de entrada estándar[26] anotando la traza de los accesos y marcándolos en el gráfico, iremos aumentando la tonalidad de la zona del acceso (zonas calientes).

A continuación se muestra el gráfico de próximos accesos de primer orden[27], resolución 50 para la carga estándar. Es decir, indica las zonas del índice que se accederán en la próxima iteración con mayor probabilidad.

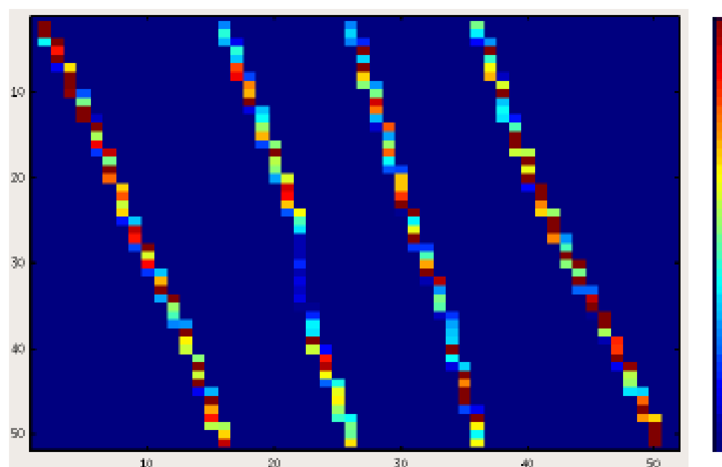

Figura C2.1 GPA primer orden (resolución 50)

[25] Resolución es la magnitud que relaciona cuantas posiciones del índice quedan agrupadas en un pixel. Una resolución de 500 índica que el gráfico ocupa 500x500 pixeles y cada uno de ellos representa |indice|/500 posiciones del índice

[26] Un millón de cadenas de entre 30 y 100 caracteres con el 100% de ellas contenidas en el índice de referencia

[27] Orden del gráfico denota la distancia máxima entre accesos que se anotan en el mismo

Podemos ver como se forman 4 rectas destacadas fruto de la acumulación de accesos en esas zonas. Lo viene a demostrar algo que de forma razonada se muestra evidente. Dado una posición inicial potencialmente solo podemos realizar 4 saltos (con A,C,G y T). De lo anterior se deriva que sean 4 las zonas de acceso para cada posición inicial. Lo que ya no es tan evidente es que estas zonas dispuestas de forma ordenada natural al índice formen 4 rectas bien definidas. No obstante, al recordar la naturaleza del método BWT (ordenación de cadenas rotadas), se puede entender el porqué de este patrón. Se trata de la expresión gráfica de la ordenación implícita del índice FM.

Esto, sin ir más lejos, nos da pie a pensar que los accesos no son totalmente aleatorios y pueden ser fácilmente modelados (y predichos). Y pese a que la densidad de accesos parece caótica dentro de las rectas de acceso de primer orden, hay que recordar que son resultados modelados a través de un experimento con un número de cadenas concreto de unas determinadas características (sujeto a desviaciones, etc). Por ellos, no debe preocuparnos a priori la densidad de los accesos en las rectas. No obstante, podemos aumentar el orden y la resolución del gráfico anterior. Si aumentamos la resolución y el orden. Es decir, mostrando un gráfico de quinto orden podremos visualizar las zonas del índice que dada una posición inicial se acceden como resultado de hasta 5 iteraciones del método. Para poder distinguir esta diferencia de orden, los accesos de menor orden tienen más peso28 que los de mayores órdenes (más alejados en la línea temporal). Además, aumentamos la resolución, reduciendo el número de anotaciones de acceso que colisionan en el espacio de posiciones del índice y pudiendo visualizar de forma más nítida los resultados.

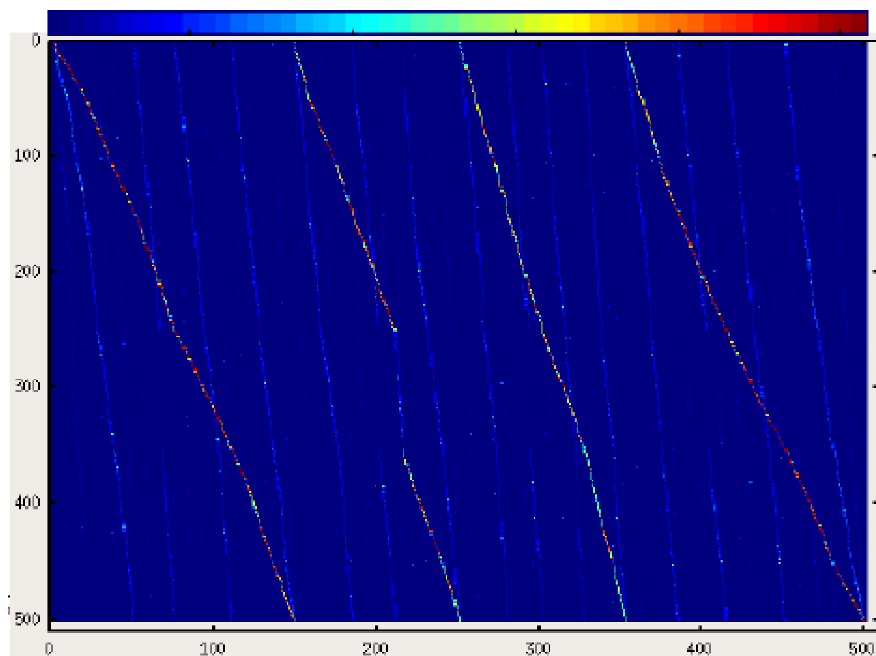

Figura C2.2 GPA quinto orden (resolución 500)

[28] Se supone una ponderación decreciente exponencial según aumenta el orden

En el gráfico anterior podemos apreciar que, además de las rectas de acceso de primer orden anteriormente mostradas, aparecen otras rectas (paralelas entre ellas) de menor intensidad. Estas rectas - a la que denominaremos de segundo, tercer y respectivo orden – modelan los accesos de su respectivo orden. De modo que los accesos a más de una iteración vista también poseen una ordenación relativa y pueden ser fácilmente modelados por una recta[29]. Además, si las rectas de primer orden eran 4, las de segundo serán 16 y, en general, 4^(orden).

Notar que solo se aprecian de forma nítida las rectas de primer y segundo orden. De mayor orden solo figuran accesos puntuales. Esto es debido a que el gráfico se ha realizado con una carga de entrada sintética y limitada. Se basa de forma empírica en los accesos efectuados por un juego de datos de entrada concreto. Por ello, podemos ir un paso más allá y plantear una exploración intensiva de las posibilidades dada una posición del índice concreta todos los "siguientes" accesos posibles con los caracteres ACGT. El resultado se muestra en el siguiente gráfico.

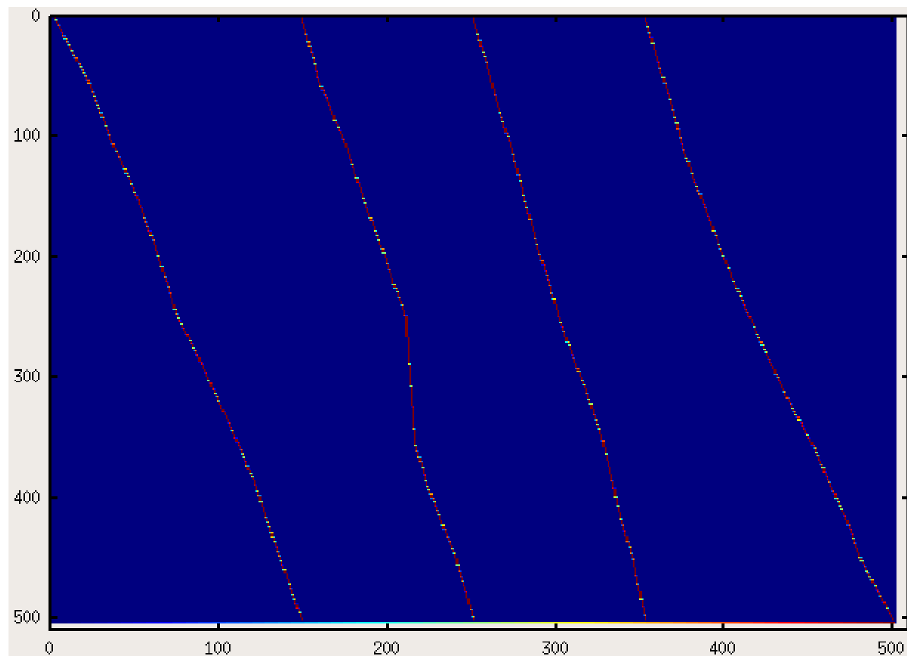

Figura C2.3 GIA primer orden (r = 500)

En el gráfico se aprecian de nuevo las rectas de acceso de primer orden. No obstante, la densidad de los accesos está basada en la equiprobabilidad de todas las cadenas y no en la experiencia con una carga de entrada concreta.

[29] Se podría razonar como en el caso anterior el porqué de este patrón en los accesos de n-ésimo orden.

Del mismo modo que en el caso anterior, si aumentamos el orden del gráfico, es decir, consideramos no solo los accesos inmediatos sino las posiciones que visitaremos en N iteraciones del bucle. Podemos ver el siguiente gráfico de donde se extraen conclusiones similares pero en el contexto de equiprobabilidad en las cadenas a buscar.

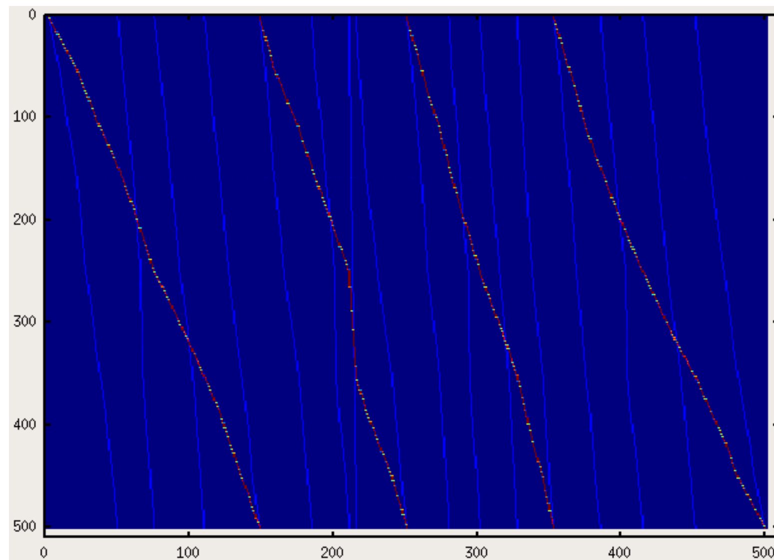

Figura C2.4 GIA tercer orden (r = 500)

Dada la cercana relación entre el número de rectas en el gráfico y las letras de nuestro alfabeto podemos plantearnos el averiguar si tiene alguna relación. Para ello, a continuación se muestra un gráfico similar a los anteriores. La diferencia radica en que los accesos al índice no "calientan" los pixeles del gráfico. En vez de eso, simplemente se colorea la casilla con el color asociado a la letra que causo el acceso[30]. De este modo, podemos trazar que accesos fuero causados por qué letras del alfabeto.

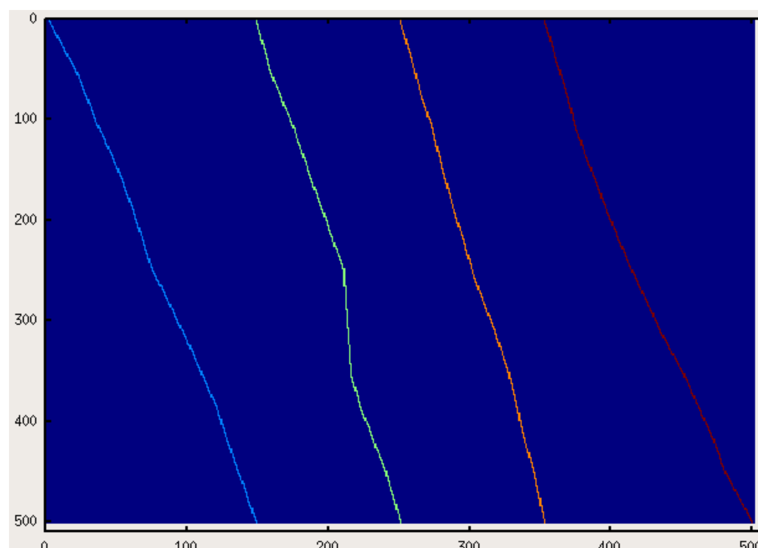

Figura C2.5 GIA-color primer orden (resolución 500)
[A=azul,C=Verde,G=naranja,T=Rojo]

[30] Si más de una letra causa el acceso en ese bloque dada una resolución, se marcará de color rojo en signo de conflicto.

Podemos observar que las cuatro rectas de primer orden están - cada una – asociadas a una letra del alfabeto[31]. Y las cuatro rectas, de izquierda a derecha, asociadas cada una a la misma letra (ordenadas alfanuméricamente) que posición del gráfico ocupan. De este modo, merece observar la situación si aumentamos el orden progresivamente.

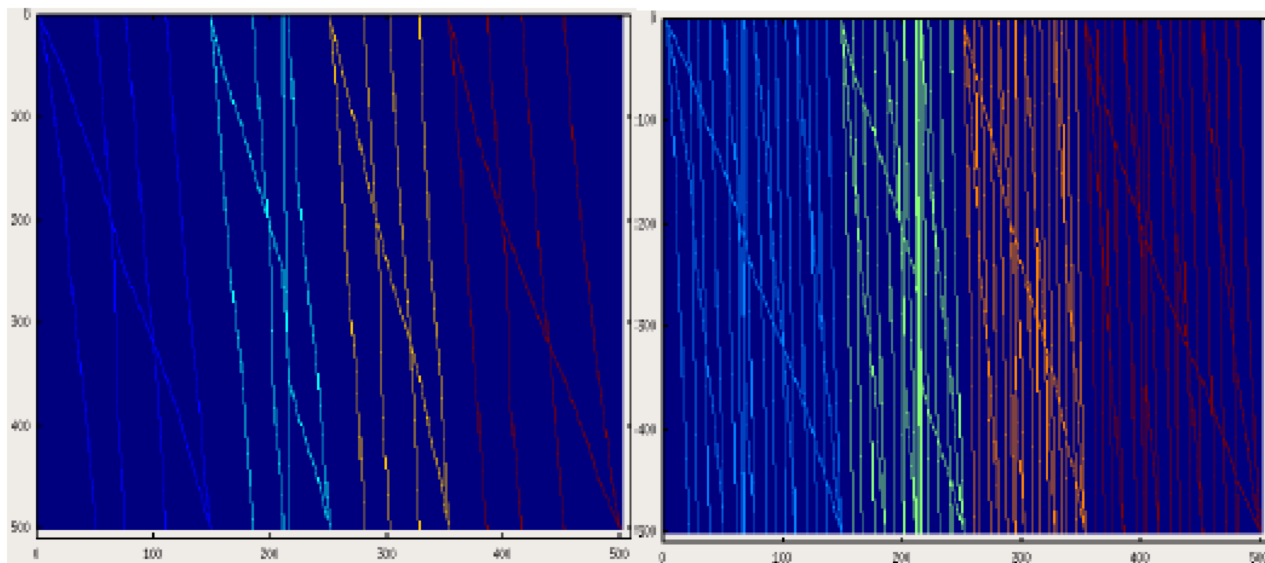

Figura C2.6 GIA-color segundo y tercer orden respectivamente (resolución 500)

Podemos ver como aparecen las rectas de accesos de segundo y tercer orden respectivamente (son 16 rectas de 2º orden y 64 de 3º). Se ve claramente como las rectas se asocian a su respectiva letra del alfabeto locamente (no se dispersan) y como no colisionan los accesos entre las diferentes letras.

Además, notar que la pendiente de estas rectas es cada vez más pronunciada. Ver que la proyección sobre las abscisas de todas las rectas asociadas a una letra de cualquier orden tiene determina un intervalo en el índice continuo y acotado.

Estos resultados los hemos encontrado muy interesantes a la hora de modelar los accesos al índice, dado que ahora sabemos que para cada posición inicial y letra solo existe 1 posición del índice final iterando en el bucle una vez. Además, podemos localizar la zona del segundo acceso conociendo la segunda letra. Y así sucesivamente. Para ilustrar este resultado podemos ver en el siguiente gráfico representadas las rectas de hasta tercer orden. En el primer gráfico hemos asociado a cada recta un color en función de la letra anterior a la que produjo el acceso. Y en el segundo, en función de la letra dos veces anterior. De esta forma, podemos trazar gráficamente a que patrón de letras corresponde cada recta.

[31] Se podría razonar de manera similar a los casos anteriores el porque de esta distribución aludiendo a la ordenación implícita del índice FM por el método del BWT

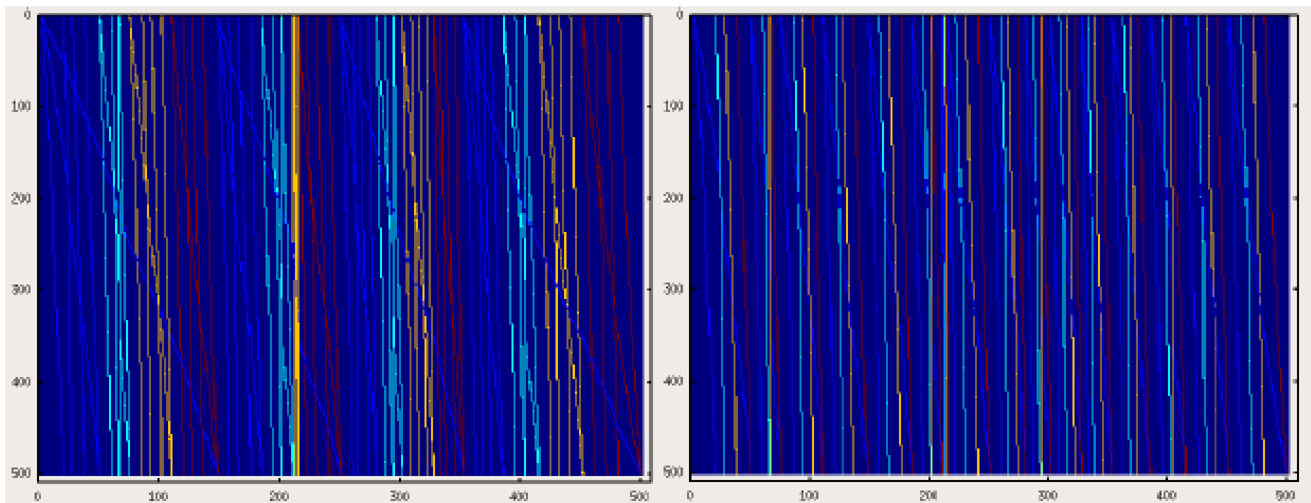

Figura C2.7 GIA-color de tercer orden donde el color se asocia con el penúltimo y antepenúltimo acceso respectivamente (resolución 500)

Se puede ver como la primera recta de orden 3 por la izquierda se asocia a la secuencia AAA, la siguiente a AAC, la siguiente a AAG, etc. De esta forma, se aprecia como las rectas corresponden a patrones de accesos. Así dada una cadena, se pueden conocer de antemano cuales son las zonas del índice que se consultarán en su búsqueda en el índice FM.

C2.1.2 Gráficos de accesos al índice ordenados para la función fm_bsearch

El análisis anterior, en busca de un orden en los accesos al índice, nos lleva a plantearnos cuales son las posiciones del índice que se acceden en una búsqueda normal. De esta forma, exponemos los gráficos de accesos al índice ordenados.

Estos gráficos muestran en el eje de las ordenadas cada iteración del bucle de fm_search (cada paso del algoritmo) y en el de las abscisas las posiciones del índice. De esta forma, se anotan los accesos realizados en cada iterativo del algoritmo "calentando" las zonas de los accesos. Así se pueden apreciar los accesos posibles que se pueden realizar en cada iteración del algoritmo[32].

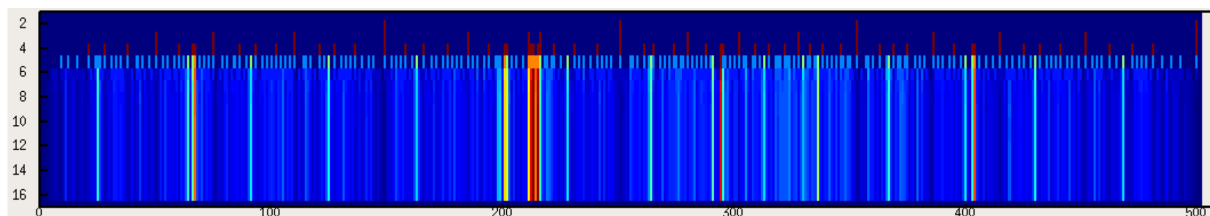

Figura C2.8 GAO-color hasta la iteración 15 (resolución 500)

En este primer gráfico se muestran la distribución de los accesos al índice en cada iteración del algoritmo profundizando hasta la iteración 15. Podemos ver la clara acumulación muy focalizada de accesos en las primeras 3 iteraciones. Además, se aprecia que las zonas calientes, a partir de la iteración 6 son fijas (o al menos a esta resolución). Esto es debido a la naturaleza del índice de referencia y revela una tendencia para las búsquedas

Al igual que en el apartado anterior, podemos mostrar el carácter causa del acceso. Asociando un color a cada letra podemos ver en los siguientes gráficos, para cada iteración del algoritmo, que partes del índice son accedidas para cada letra del alfabeto.

En la primera, no damos importancia en la intensidad del color al orden de los accesos. Claramente se pueden apreciar las zonas del índice asociadas a cada letra y la concentración en pocas posiciones de los primeros accesos. En la segunda, se ha asociado intensidad a los accesos (a más accesos mas intenso) y el color rojo cuando se producen más accesos que los que la resolución nos permite visualizar.

Las dos últimas gráficas son resultado de la descomposición de la primera según los accesos se han iniciado desde la última posición del índice (HI) o la primera (LO).

[32] Exploración intensiva de las posibilidades

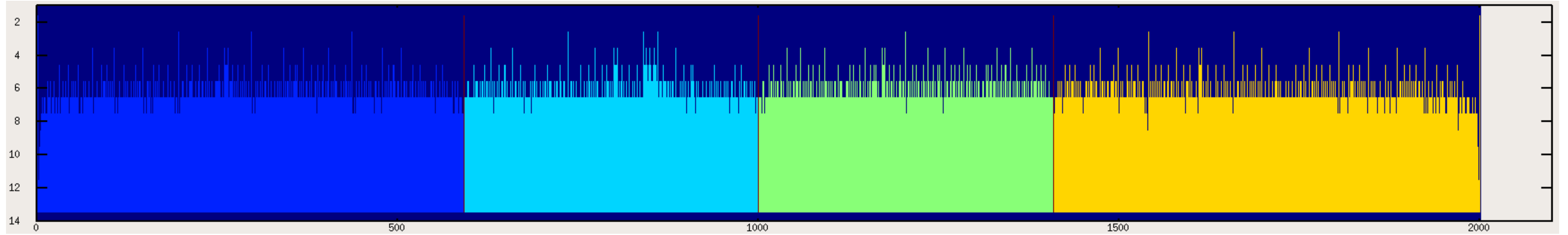

Figura C2.9 GAO-color hasta la iteración 12 (resolución 500)

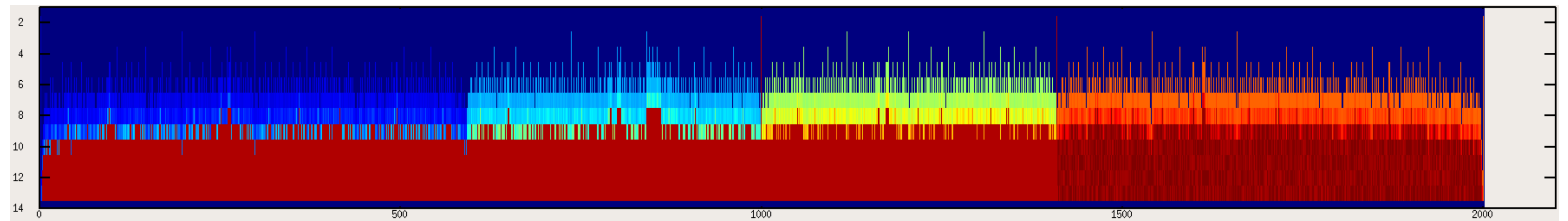

Figura C2.10 GAO-color con zonas calientes hasta la iteración 12 (resolución 500)

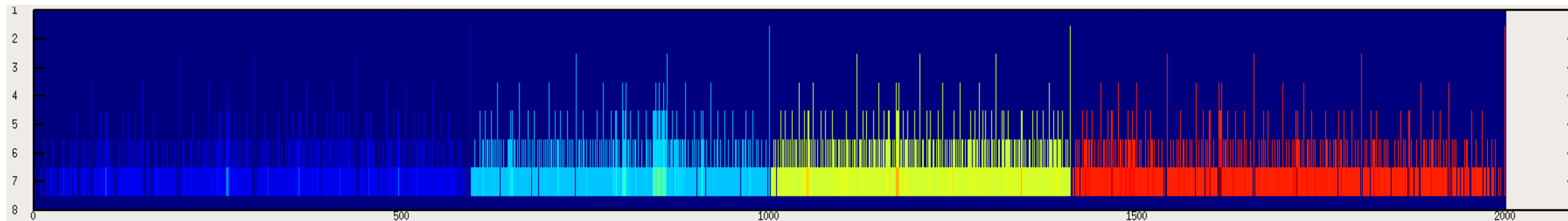

Figura C2.11 GAO-color desde HI hasta la iteración 6 (resolución 500)

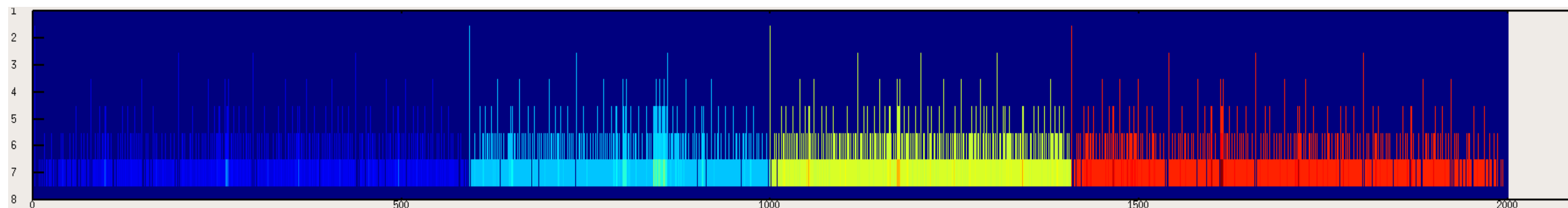

Figura C2.12 GAO-color desde LO hasta la iteración 6 (resolución 500)

C2.1.3 Gráficos de accesos al índice por cadenas para fm_bsearch

Por otro lado, nos interesa la información de los gráficos anteriores pero para cadenas concretas. Esto nos mostrará información relacionada con los patrones de acceso al índice para cadenas de búsqueda concretas. Vamos a definir un conjunto pequeño de cadenas lo suficientemente pequeñas como para poder analizar fácilmente los resultados. Además, hemos procurado que la entropia de las cadenas sea alta para obtener resultados con accesos dispersos en el índice (a la vista de los resultados anteriores).

Recordar que hi y lo son los dos centinelas encargados de acotar el intervalo en el índice donde se hallan las cadenas que hasta la iteración actual coinciden con la cadena a buscar[33].

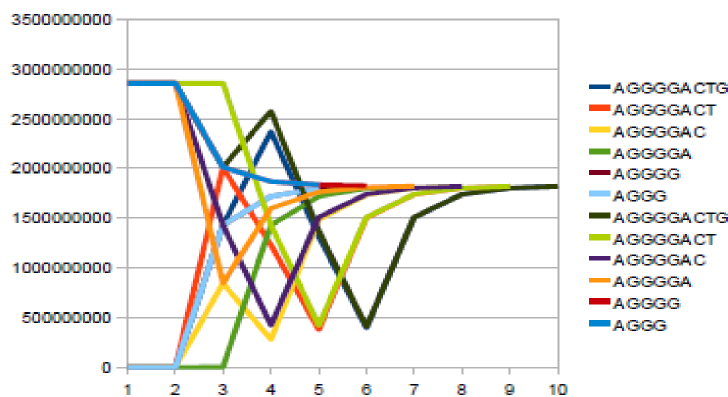

Figura C2.13 Accesos al índice desde hi y lo

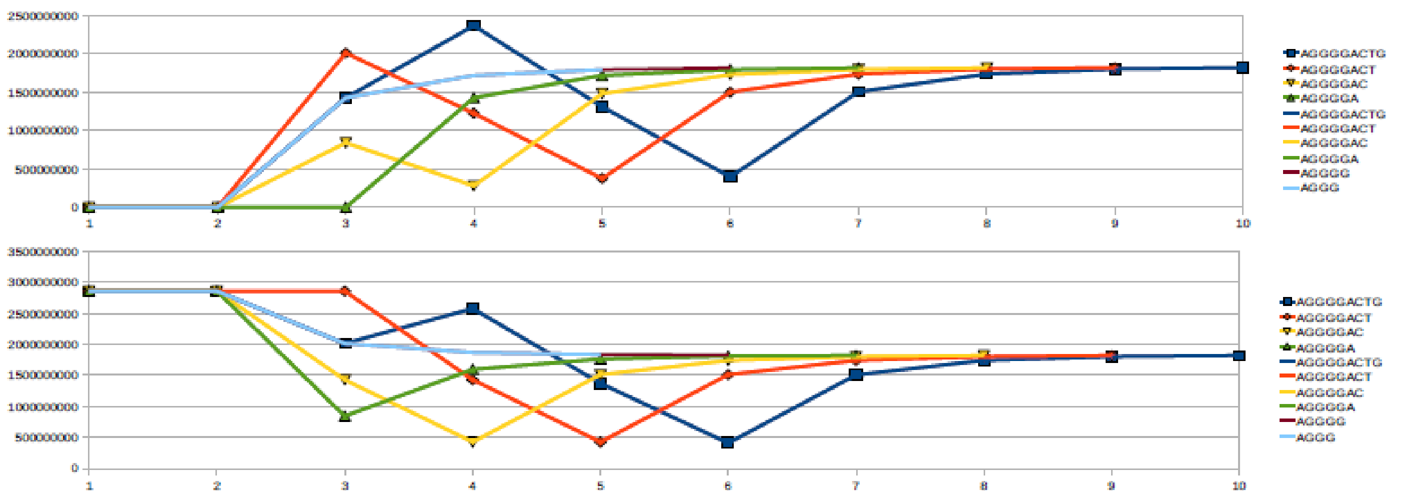

Figura C2.14 Accesos al índice desde hi y lo respectivamente

[33] Las cadenas entre hi y lo del índice comparten el mismo sufijo que la cadena buscada desde n-iter hasta n

Podemos ver como se realizan accesos múltiples a las mismas posiciones del índice (se repiten asiduamente). Notar que debido a la naturaleza de la búsqueda hacia atrás, cadenas con el mismo posfijo realizan los mismos accesos durante los caracteres comunes del mismo. Se puede ver en la siguiente gráfica como influiría la ordenación de los accesos para evitar accesos repetidos.

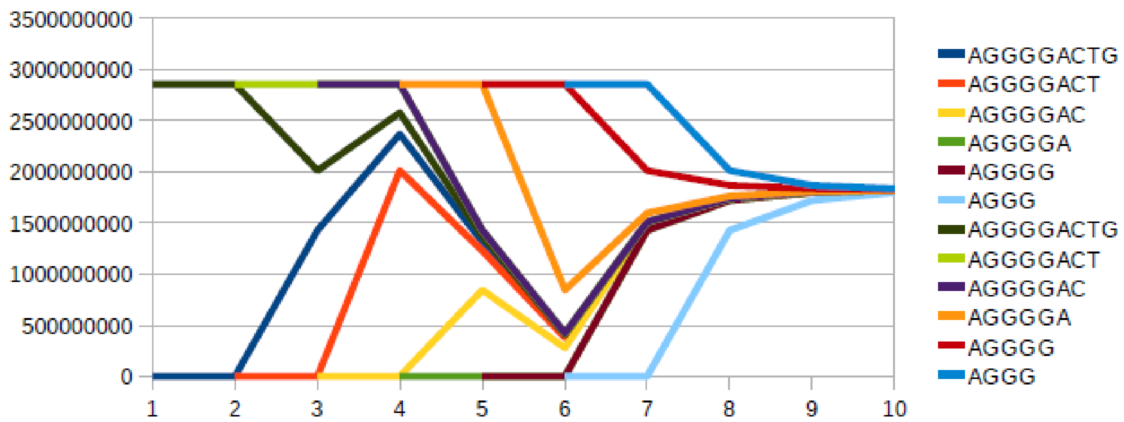

Figura C2.15 Accesos al índice desde hi y lo [ORDENADOS]

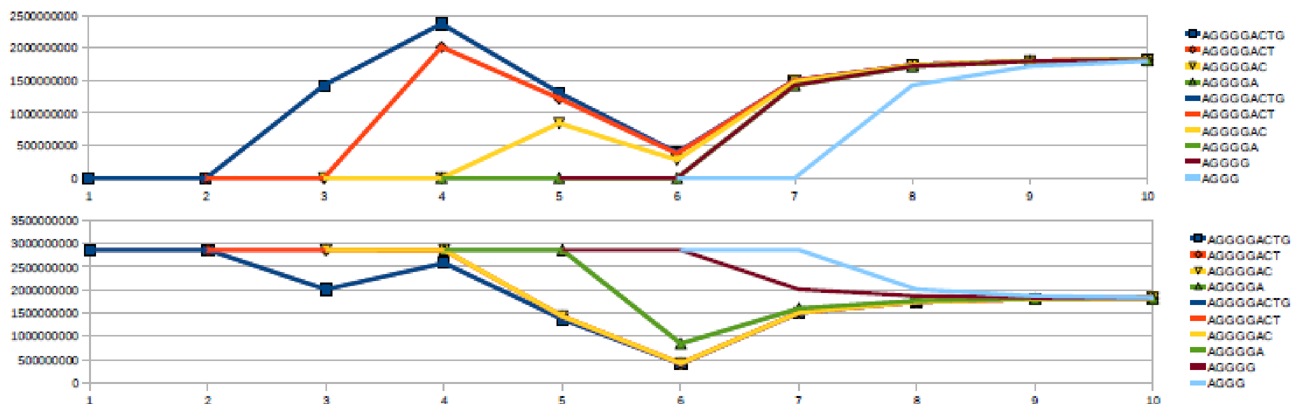

Figura C2.16 Accesos al índice desde hi y lo respectivamente [ORDENADOS]

Además, conviene observar el comportamiento en los accesos al índice en la búsqueda de un conjunto de cadenas con una entropia muy baja. Ver que los accesos pueden variar la zona del índice de exploración en cualquier momento. No obstante en “runs” de caracteres la búsqueda se focaliza en una sola zona.

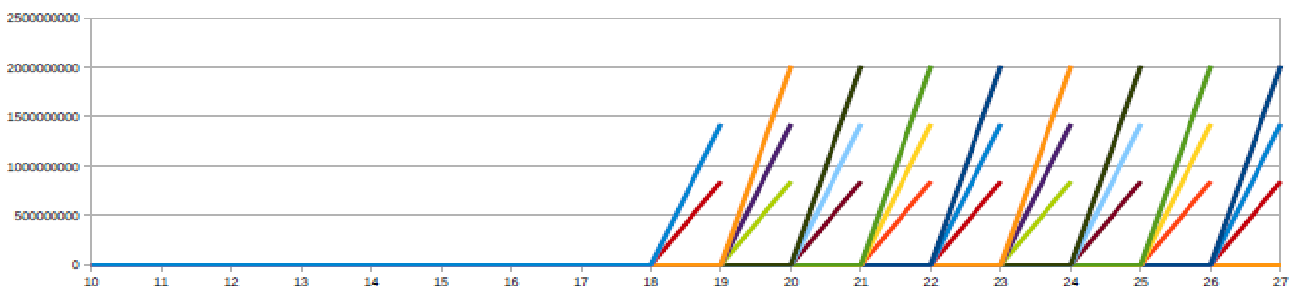

Figura C2.17 Accesos al índice desde lo para A...A{A,C,G,T}^n

Por otro lado, destacar que para una cadena de entre 50 y 100 la unicidad de su existencia en el índice de referencia esta casi garantizada. Esto se puede intuir de los gráficos de acceso al índice ordenados. Esto tiene la implicación de que para toda cadena única en el índice, a partir de cierta iteración, se cumple el invariante (hi-lo=1). A continuación se muestran las figuras asociadas.

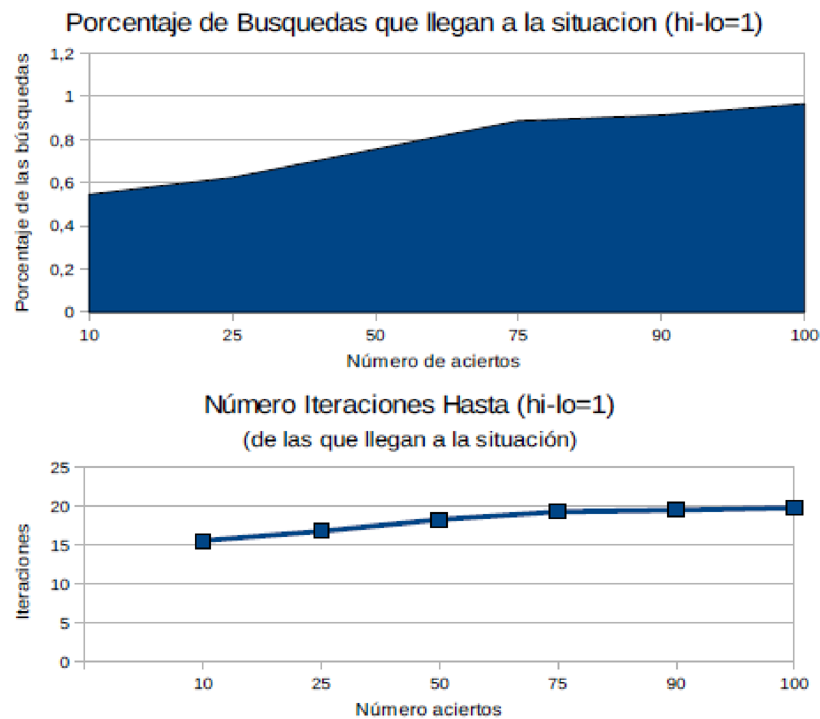

Figura C2.18 Gráficos sobre el comportamiento de los centinelas hi y lo

Estos resultados son relevantes en tanto y cuanto que cuando hi-lo=1 el intervalo de cadenas candidatas es 1. Por ello, en siguientes iteraciones hi y lo realizan accesos similares sobre la misma parte del índice. Sabiendo esto y teniendo presente el método de búsqueda en índices FM de la literatura se pude plantear una alternativa de mejora. Esta alternativa se describe en el apartado 7 y consiste básicamente en unificar las dos llamadas de hi y lo en una sola en el momento que hi-lo sea 1.

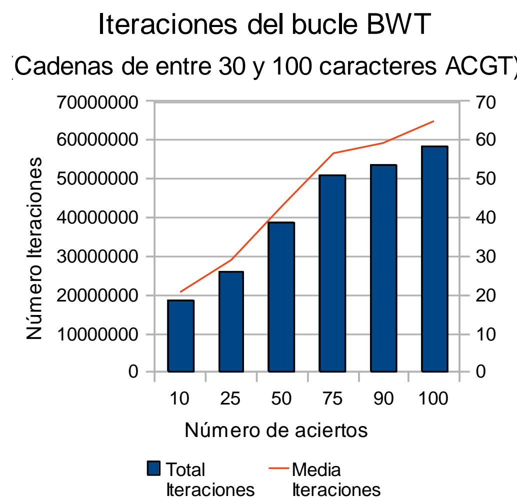

Figura C2.19 Iteraciones del bucle principal de fm_bsearch

C2.2 Análisis de los accesos a memoria de la función fm_lookup

De la misma forma que se ha hecho con fm_bsearch, se han analizado las posiciones del índice accedidas por la función fm_lookup para la conversión de posiciones en el indice FM a posiciones en el texto original.

```
entero32 fm_lookup(FM indiceFM, entero32 posicionIndiceFM) {
  entero32 distancia = 0, i = posicionIndiceFM;
  while ( La posicion i no esta almacenada explicitamente ) {
    i=_BWT_LF(indiceFM.indiceBWT,i);
    distancia++;
  }
  return (decodificarPosicionFM(i)+distancia);
}
```

Figura C2.20 Esquema algorítmico de fm_lookup

Como ya se ha comentado, la función fm_lookup se encarga de mapear las posiciones del índice FM en las del texto original. Su mecánica es bastante simple. El algoritmo recorre hacia atrás el texto del indice Fm desde la posición pasada por parámetro[34]. Cada vez que realiza esto comprueba si la nueva posición tiene su traducción almacenada en el índice FM explícitamente. Una vez encontrado un anclaje[35] (posición marcada) la posición correspondiente al índice proporcionado es la suma de la posición correspondiente al ancla más la distancia recorrida hacia atrás.

[34] Posición que apunta a una cadena en el índice FM. Posiblemente devuelta por fm_bsearch

[35] Nos referimos como ancla o posición marcada a aquellas posiciones del índice cuya correspondiente en el texto original se han almacenado explícitamente.

C2.2.1 Gráfico de accesos al índice para la función fm_lookup

Dado que la forma de recorrer el texto hacia atrás es llamando a "bwt_dna_LF", la función fm_lookup comparte muchas similitudes con fm_bsearch en la forma en la que accede al índice. No obstante, fm_lookup no proporciona ningún carácter a la función, sino que se basa en el carácter que esta en la posición proporcionada para realizar el Last-to-First Step (LF).

Vamos a generar un gráfico donde vamos marcando las posiciones del índice que son accedidas por fm_lookup para decodificar una posición dada. Es decir, para cada ordenada (como índice que queremos decodificar) marcaremos en su fila las ordenadas a las que se accede con fm_lookup.

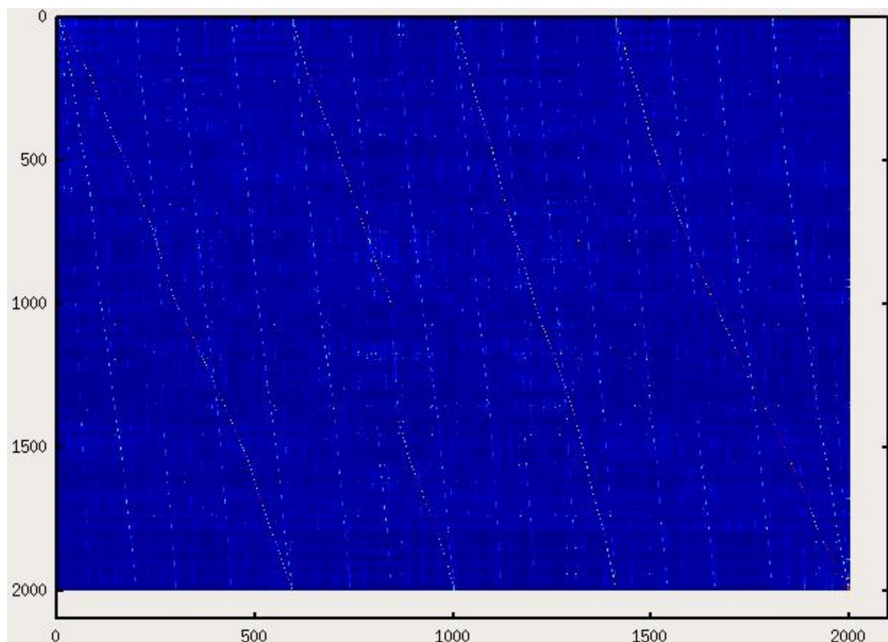

Figura C2.21 Gráfico de accesos de fm_lookup (Resolución 2000)

Podemos ver que los accesos marcan un patrón similar a los obtenidos con fm_bsearch. Esto viene a corroborar de forma gráfica lo que ya habíamos derivado del algoritmo. Dado que tanto fm_bsearch como fm_lookup se basan en el BackwardSearch como paso básico de recorrido del índice comparten los mismos patrones en el acceso al mismo.

C2.2.2 Análisis de los anclajes para la función fm_lookup

Dadas las similitudes en los accesos con fm_bsearch, centraremos nuestra atención en otro punto clave: la distribución de los anclajes en el índice.

Sabemos por la teoría que si la distribución de anclas en el índice Fm es uniforme con respecto al texto original, podemos acotar el número de iteraciones del bucle principal de fm_lookup por la distancia máxima entre dos anclajes (igual para todas las posiciones). Ademas sabemos que el valor usado en el índice de referencia para la distancia entre dos anclajes (sampling_rate) es 32. No obstante, los anclajes han sido repartidos de forma uniforme sobre el índice FM, no con respecto al texto original. Esto implica que ya no disponemos de la cota superior mencionada y que algunas ejecuciones pueden iterar por debajo de 32 y otras muy por encima de forma no controlada.

Vamos a explicar más en detalle esta idea pues es importante entender este punto. Actualmente, el algoritmo de fm_lookup determina si se ha alanzado un anclaje realizando una operación de modulo con sampling_rate. Esto es debido a que los anclajes se han distribuido de forma homogénea respecto al índice FM. Cada iteración del bucle retrocede una posición en el texto original en busca de un anclaje. No obstante, lo que se recorre de manera lineal es el texto original y no el índice FM (el cual ya ha demostrado se organiza de forma muy particular con respecto al texto original). Como consecuencia de esto, una ejecución de fm_lookup puede iterar un número no determinado de veces. En el caso teórico peor |Texto| - 32.

Veamos un sencillo caso donde este resultado se puede apreciar de forma clara. Hemos buscado tres cadenas con sufijo común. En teoría ninguna de las tres búsquedas debería exceder las 32 iteraciones. No obstante solo una de ellas está por debajo de las 32. Las otras dos cadenas iteran 40 y 89 veces.

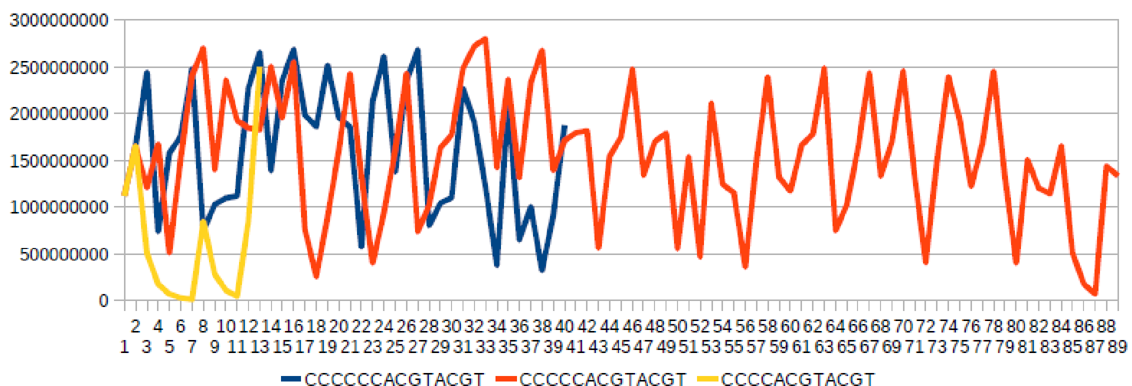

Figura C2.22 Gráfico de accesos al índice en cada iteración de fm_lookup para 3 cadenas

Cada iteración que se realiza es un acceso completo al índice con la consecuente descompresión del carácter que ocupa esa posición. Sin duda es un coste que conviene explorar la posibilidad de mejorar.

Si marcamos todos los accesos que realiza fm_lookup para todas las posiciones del índice FM, obtenemos el siguiente gráfico.

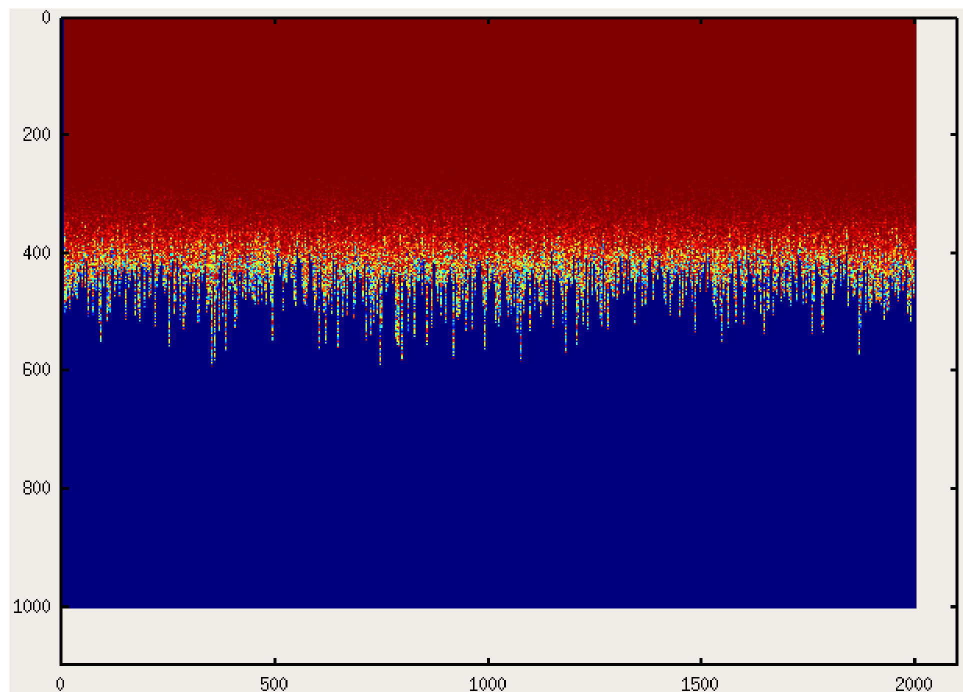

Figura C2.23 Gráfico de accesos al índice por cada iteración para la decodificación intensiva con fm_lookup de todas las posiciones del índice FM

En el eje de las abscisas se han representado las posiciones del índice FM accedidas con una resolución de 2000. En las ordenadas el número de iteración de fm_lookup (desde 0, la primera, hasta una hipotética 1000). Se ha realizado la decodificación de todas las posiciones del índice. Es decir, se ha llamado a fm_lookup con todas posiciones posibles del índice. Entonces para cada iteración se ha marcado el acceso realizado por cada búsqueda de fm_lookup. De esta forma, se marcan las zonas calientes y se muestra el gráfico anterior.

Se puede observar como el área explorada se extiende por la totalidad del índice hasta las 400 iteraciones con una densidad de accesos es muy elevada. Incluso se pueden observar picos de 600 iteraciones (luego hay llamadas a fm_lookup que iteran 600 veces).

En en marco teórico, el área roja no debería sobrepasar el nivel de las 32 iteraciones. Y, por supuesto, ocupando el mismo espacio que ocupan ahora el índice de anclas.

Incluso, para destacar esta idea, vamos a representar los accesos que realizan para decodificar cada posición del índice. Se ha dividido las posiciones del índice a buscar en 2000 partes (resolución) y representado cuantos accesos al índice generan cada bloque.

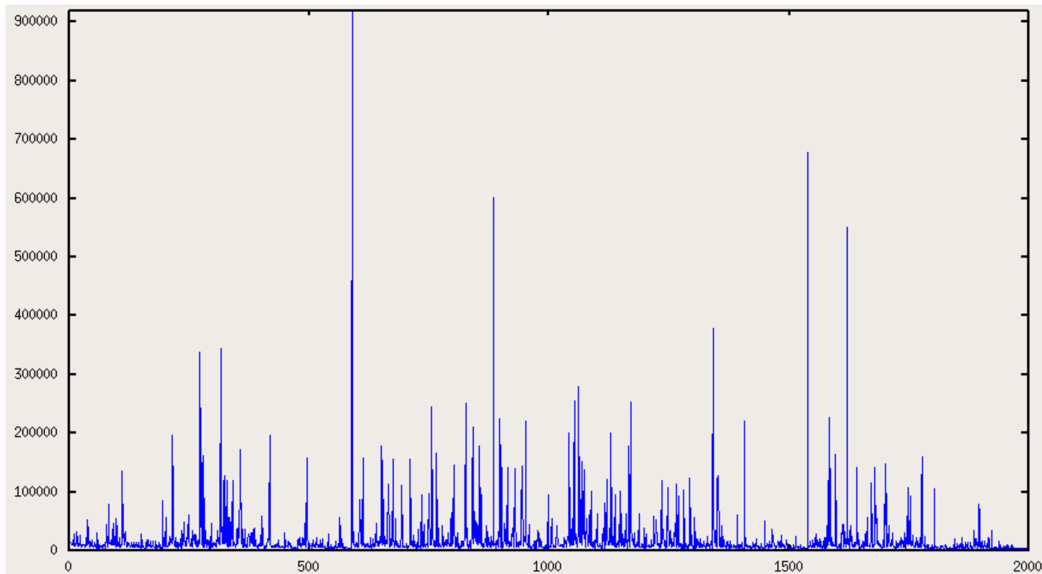

Figura C2.24 Gráfico de accesos al índice para la decodificación de 2000 bloques de posiciones del índice FM

Notar como el número de accesos para los bloques no es homogéneo y causa que algunos bloques realicen un número muy elevado de accesos.

D. Análisis de propuestas en la construcción de indices FM e implementación de kernels

D1. Propuestas a la organización a bajo nivel del índice de DNA

Aparte de las alternativas presentadas en la primera parte de la memoria se han implementado una serie de índices alternativos para evaluar su rendimiento y desempeño. La idea que se ha perseguido en este proceso ha sido la de simplificar al máximo la estructura para que las operaciones del kernel fueran ligeras.

Uno de los principales cambios ha sido la descomposición del bitmap, que codificaba las bases todas juntas, en cuatro bitmaps separados donde se codificaran las letras por separado. Esta mejora elimina las operaciones de extracción y conteo de los caracteres en el bitmap por una simple operación de popCount en el bitmap indicado. Solo con esta alternativa se ha podido obtener un speedup de 1,5x.

Por otro lado, se ha planteado eliminar el doble bucketing de la organización inicial. Dado que con dos niveles el número de operaciones básicas en el kernel se multiplica por un factor de 2,3. Con solo un nivel el número de operaciones básicas necesarias para acceder al índice se reduce drásticamente. No obstante tiene el inconveniente de que el índice ocupa más espacio en memoria, hasta un factor de 1,1 sobre el texto original.

Por último, se ha variado el tamaño del bitmap (o bitmaps) que codificaban las bases. Esta alternativa pretende explotar el uso de aceleradores en los kernels y ahorar un gran número de operaciones de popCount. Recordar que debido a las propiedades de la transformada BWT, el texto resultante tiene a agruparse en secuencias homogéneas de bases. Por ello, a menor tamaño de bitmap mayor es la probabilidad de codificar una secuencia de una sola base. Así pues, los aceleradores pueden suponer hasta un 8% sobre el rendimiento global por el número de operaciones que ahorran ejecutar en los kernels.

D2. Propuestas a la implementación de los kernels

Para cada uno de los índices planteados se ha implementado su correspondiente kernel. En este proceso se plantearon varias alternativas para comparar como pequeñas variaciones en las implementaciones de los kernel pueden afectar al rendimiento global de la librería.

Una de las cuestiones que se planteó es la de incluir la implementación de la detección de los aceleradores en los kernel. Recordar que un acelerador es información extra contenida en el índice con el fin de evitar tener que realizar la operación de popCount. La aceleración se puede hacer en varias fases del acceso al índice y dependiendo de esto se ahorran más o menos operaciones del acceso. De la implementación y evaluación de las propuestas se concluye que merece la pena implementar todos los aceleradores dado el gran coste extra de realizar operaciones de popCount.

Otras cuestiones también fueron consideradas. Como la de desplegar el tratamiento para cada carácter en particular o la reutilización de variables en el acceso. Se puedo ver como las diferencias eran mínimas y el compilador no es sensible a este tipo de optimizaciones.

D3. Propuestas a la implementación de la operación popCount

A continuación se muestran una serie de implementaciones alternativas de la operación popCount. Recordemos que este operador cuenta el número de unos dentro de una palabra. Esta operación está en el camino crítico de la librería GEM y es de suma relevancia su desempeño para el rendimiento global de la aplicación.

```
inline int bitcount_iteratedCount(unsigned int n) {
  int count = 0;
  while (n) {
    count += n & 0x1u;
    n >>= 1;
  }
  return count;
}
```

Figura D3.1 PopCount iterativo

```
inline int bitcount_sparceOnes(unsigned int n)  {
  int count = 0 ;
  while (n)  {
    count++ ;
    n &= (n - 1) ;
  }
  return count ;
}
```

Figura D3.2 PopCount para palabras poco densas en unos

```
inline int bitcount_denseOnes(unsigned int n)   {
  int count = 8 * sizeof(int) ;
  n ^= (unsigned int) - 1 ;
  while (n)  {
    count-- ;
    n &= (n - 1) ;
  }
  return count ;
}
```

Figura D3.3 PopCount para palabras densas en unos

```
#define TWOPC(c)     (0x1u << (c))
#define MASKPC(c) \
  (((unsigned int)(-1)) / (TWOPC(TWOPC(c)) + 1u))
#define COUNT(x,c) \
  ((x) & MASKPC(c)) + (((x) >> (TWOPC(c))) & MASKPC(c))

inline int bitcount_parallelCount(unsigned int n)  {
   n = COUNT(n, 0) ;
   n = COUNT(n, 1) ;
   n = COUNT(n, 2) ;
   n = COUNT(n, 3) ;
   n = COUNT(n, 4) ;
   return n ;
}
```

Figura D3.4 PopCount por cuenta paralela

```
#define MASK_01010101 (((unsigned int)(-1))/3)
#define MASK_00110011 (((unsigned int)(-1))/5)
#define MASK_00001111 (((unsigned int)(-1))/17)

inline int bitcount_niftyParallelCount(unsigned int n) {
   n = (n & MASK_01010101) + ((n >> 1) & MASK_01010101) ;
   n = (n & MASK_00110011) + ((n >> 2) & MASK_00110011) ;
   n = (n & MASK_00001111) + ((n >> 4) & MASK_00001111) ;
   return n % 255 ;
}
```

Figura D3.5 PopCount por desplazamiento y suma

```
inline int bitcount_MIT_HAKMEM_Count(unsigned int n) {
    register unsigned int tmp;
    tmp = n - ((n >> 1) & 033333333333)
            - ((n >> 2) & 011111111111);
    return ((tmp + (tmp >> 3)) & 030707070707) % 63;
}
```

Figura D3.6 PopCount por el algoritmo de HAKMEN del MIT

De este modo se han evaluado las diferentes implementaciones junto a la extensión __builtin_popcount de gcc y la intrínseca SIMD de Intel SSE4.2 _mm_popcnt_u3 (extensión vectorial). La tabla D3.1 muestra los resultados para las diferentes implementaciones. Podemos ver como la extensión vectorial de Intel es, evidentemente, la más rápida. Dado que se ejecuta como instrucción única especializada a nivel de arquitectura obtiene el mejor rendimiento. No obstante, es destacable el rendimiento de la implementación de gcc a bajo nivel. Este ha sido optimizada para las diferentes arquitecturas y por ello obtiene buen tiempo de ejecución.

Función	Tiempo (s)
__builtin_popcount	4,21
_mm_popcnt_u32	3,77
bitcount_iteratedCount	5,34
bitcount_sparceOnes	4,75
bitcount_denseOnes	5,64
bitcount_parallelCount	4,23
bitcount_niftyParallelCount	4,84
bitcount_MIT_HAKMEM_Count	4,22

Tabla D3.1 Resultados de la implementaciones de PopCount

D4. Propuestas a la organización de CSA

Para la implementación del CSA se propusieron varias alternativas con el fin de encontrar la que mejores resultados obtuviera. A continuación se muestran las cifras relacionadas con las tres estrategias de marcado propuestas en la literatura.

Marcado Inverso

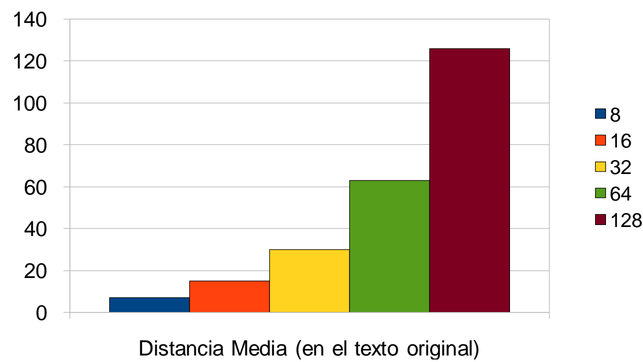

Figura D4.1 Distancia media entre anclajes

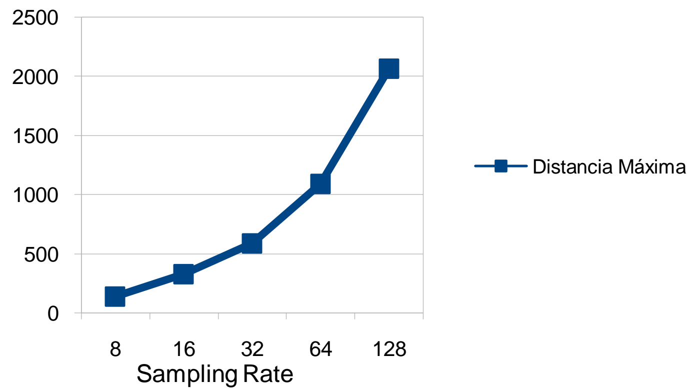

Figura D4.2 Distancia máxima entre marcas

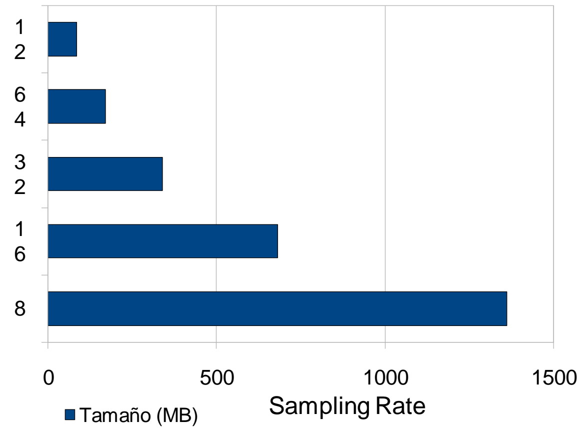

Figura D4.3 Tamaño del CSA para el marcado inverso

Marcado por letra

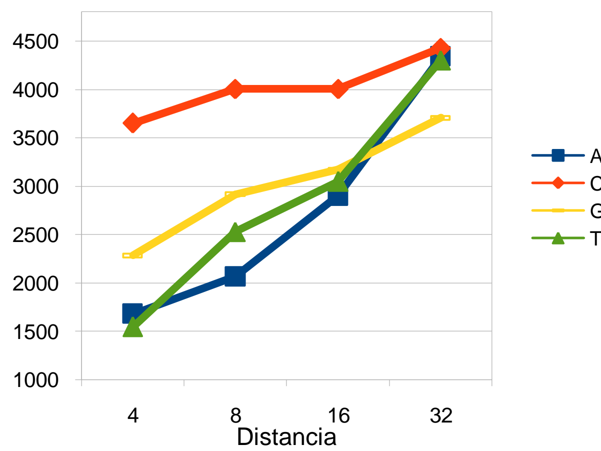

Figura D4.4 Número máximo de accesos para el marcado por letra

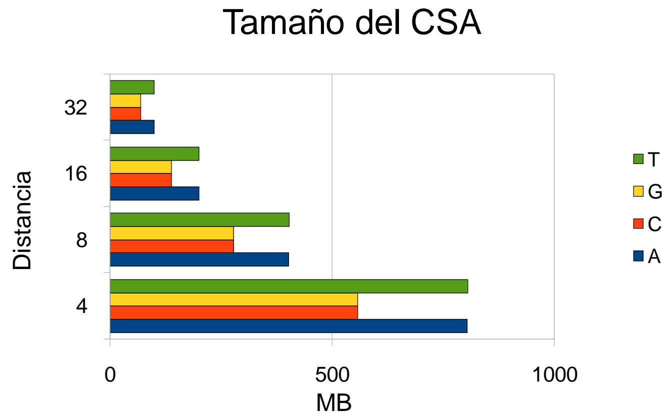

Figura D4.5 Tamaño del CSA para el marcado por letra

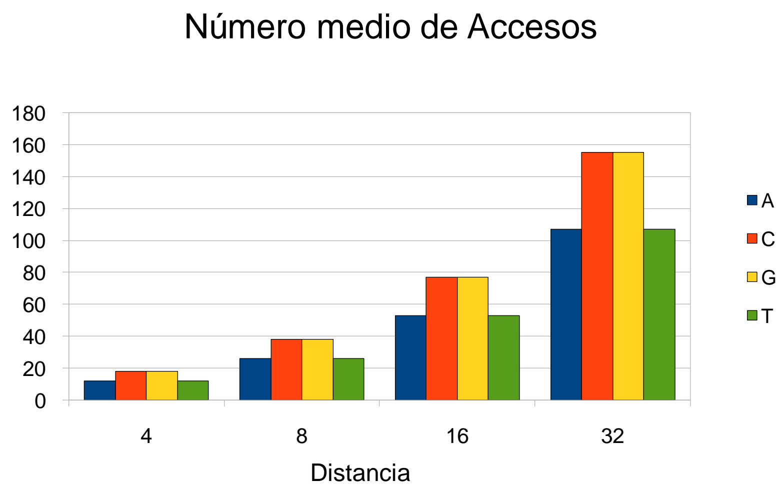

Figura D4.6 Número medio de accesos para el marcado por letra

D5. Dimensionado de la tabla de lookups para memoization

Para escoger la mejor alternativa de tabla de lookup se propusieron varias alternativas a la estructura de la tabla y su dimensión. En la figura D5.1 se pueden ver los speedUp calculados para cada una de las alternativas. Podemos ver como la mejor alternativa es la B2 para un tamaño de tabla de lookup moderado que no consuma mucha memoria.

	Tamaño del último nivel de la tabla de lookup				
Estrategia	*4*	*6*	*8*	*10*	*12*
B1	1	1,01	1,01	1,01	1,02
B2	1	1	1,07	1,14	1,17
B3	0,97	1	1,07	1,09	1,13
B4	0,98	1	1	1,01	1,01

Figura D5.1 Speedup para diferentes estructuras y tamaños de la tabla de lookup

B1: Lookup para todos los niveles de búsqueda. Llenado según se resuelven las consultas

B2: Lookup para el último nivel de la búsqueda. Llenado completo en la inicialización del índice

B3: Lookup para todos los niveles de búsqueda. Estrategia de consulta optimista

B4: Lookup para todos los niveles de búsqueda. Estrategia de consulta pesimista

D6. Dimensionado del número de queries entrelazadas

Con el fin de escoger los mejores parámetros de optimización se ejecutaron varias pruebas para dimensionar estos parámetros. En la figura D6.1 se pueden ver los resultados, siendo 20 un número adecuado de queries entrelazadas.

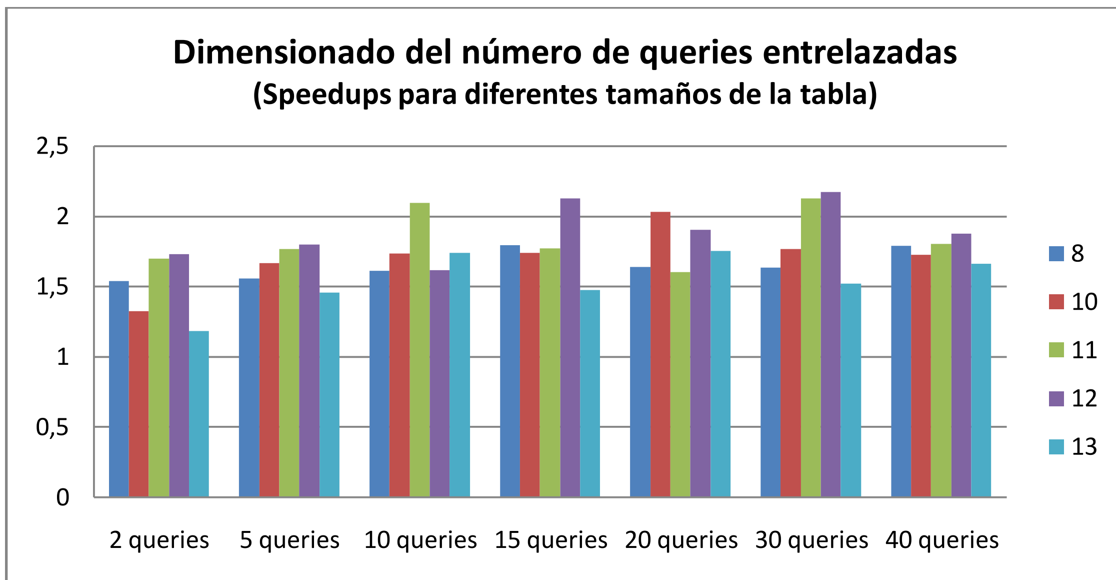

Figura D6.1 Dimensionado del número de queries entrelazadas

E. Caracterización de la organización del índice genómico de referencia

Con la finalidad de encontrar distribuciones del índice que proporcionaran facilidades para la extraccón de información y búsqueda se ha realizado un análisis del índice de referencia. El objetivo es obtener datos sobre la distribución de las bases, repeticiones, etc. Para ello se han implementado varias aplicaciones que escanean el índice y toman datos sobre la organización del mismo.

Uno de los datos analizados es la repetición de secuencias en el mismo. El objetivo es ver que las cadenas dentro del índice se repiten asiduamente y que, por ello, aplicar técnica de compresión a la búsqueda de cadenas podría ser muy beneficioso para el rendimiento final.

#	Diferent sequences	Repeted sequences	Increment	Max Repeated sequence	Diference between repeated and unique	%
30	1.325.380.917	1.532.664.995	0	101026	207.284.078	7,25%
60	1.408.067.410	1.449.978.472	82.686.493	4686	41.911.062	1,47%
80	1.418.481.248	1.439.564.614	10.413.838	2709	21.083.366	0,74%
90	1.420.829.565	1.437.216.287	2.348.317	2344	16.386.722	0,57%
100	1.422.477.289	1.435.568.553	1.647.724	2004	13.091.264	0,46%
110	1.423.714.713	1.434.331.119	1.237.424	1561	10.616.406	0,37%
120	1.424.669.519	1.433.376.303	954.806	1377	8.706.784	0,30%
130	1.425.424.221	1.432.621.591	754.702	1133	7.197.370	0,25%

Tabla E.1 Repetición de secuencias en el índice de referencia

Podemos ver que pese a variar el tamaño de las secuencias tomadas la mitad del índice son cadenas repetidas. Además observamos como el número máximo de veces que se repite una secuencia ronda las mil repeticiones. Esto ha de tenerse en cuenta dado que la decodificación de una búsqueda podría generar hasta 101.026 resultados. En la figura E.1 se representa de forma gráfica la curva de cadenas distintas frente a la de posibles combinaciones para una longitud de secuencia dada.

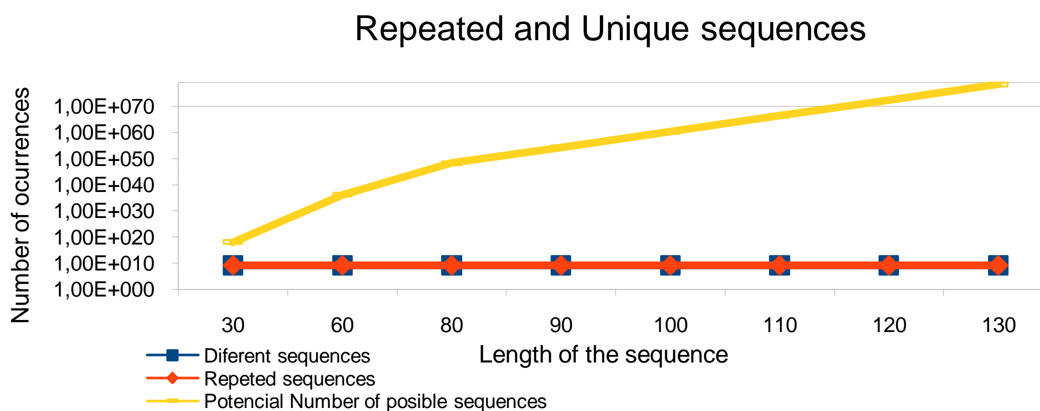

Figura E.1 Secuencias únicas frente a potenciales combinaciones de bases

Siguiendo el razonamiento, buscamos datos sobre la distribución de las cadenas en términos de conocer cuantas veces están repetidas las cadenas que no son únicas. Podemos ver en la tabla E.2 un análisis de estos datos.

#	1	10	100	1.000	10.000	100.000
30	1,28E+09	40199983	3386505	257018	27620	661
60	1,39E+09	19251252	871644	61434	5164	0
80	1,4E+09	13114007	392985	38219	1785	0
90	1,41E+09	11471894	299318	33895	1139	0
100	1,41E+09	10320122	244998	29999	697	0
110	1,41E+09	9449112	207729	27025	315	0
120	1,42E+09	8745967	180395	23879	210	0
130	1,42E+09	8159585	159486	21153	51	0
%						
30	96,69%	3,03%	0,26%	0,02%	0,00%	0,00%
60	98,57%	1,37%	0,06%	0,00%	0,00%	0,00%
80	99,04%	0,92%	0,03%	0,00%	0,00%	0,00%
90	99,17%	0,81%	0,02%	0,00%	0,00%	0,00%
100	99,26%	0,73%	0,02%	0,00%	0,00%	0,00%
110	99,32%	0,66%	0,01%	0,00%	0,00%	0,00%
120	99,37%	0,61%	0,01%	0,00%	0,00%	0,00%
130	99,41%	0,57%	0,01%	0,00%	0,00%	0,00%
LOAD						
30	29,90253	28,75465	201,8838	1477,237	14964,96	26594,52
60	30,55517	11,92364	44,02249	370,4684	1880,241	0
80	30,70378	7,644883	20,17123	244,6595	586,3429	0
90	30,74251	6,53586	15,73688	205,0647	344,8812	0
100	30,76906	5,770021	13,17053	171,4885	194,9026	0
110	30,78926	5,197076	11,28793	145,5797	85,52385	0
120	30,80541	4,741524	9,918549	119,3956	50,45433	0
130	30,81881	4,368699	8,838459	99,66854	11,7008	0
% LD Acum						
30	0,07%	0,14%	0,60%	4,01%	38,58%	100,00%
60	1,31%	1,82%	3,70%	19,55%	100,00%	100,00%
80	3,45%	4,31%	6,58%	34,08%	100,00%	100,00%
90	5,10%	6,18%	8,79%	42,80%	100,00%	100,00%
100	7,39%	8,78%	11,95%	53,16%	100,00%	100,00%
110	11,06%	12,93%	16,98%	69,28%	100,00%	100,00%
120	14,31%	16,51%	21,12%	76,57%	100,00%	100,00%
130	19,83%	22,64%	28,33%	92,47%	100,00%	100,00%
GROUP 1	**Load 1**	**% 1**	**Load > 1**	**% >1**		
30	29,90253	0,07%	43267,36	99,93%		
60	30,55517	1,31%	2306,656	98,69%		
80	30,70378	3,45%	858,8185	96,55%		
90	30,74251	5,10%	572,2187	94,90%		
100	30,76906	7,39%	385,3317	92,61%		
110	30,78926	11,06%	247,5885	88,94%		
120	30,80541	14,31%	184,51	85,69%		
130	30,81881	19,83%	124,5765	80,17%		

Tabla E.2 Datos sobre la repetición de secuencias no únicas y la carga que representan para GEM

Podemos observar que cerca del 99% de cadenas tienen menos de 10 repeticiones en el índice. No obstante, el 1% restante tiene una cifra muy elevada de repeticiones. Esto deriva en que este 1% de las cadenas supone un 99% de la carga para fm_lookup en la decodificación de cadenas. Este tipo de distribución atiende a un modelo de red de mundo pequeño. Las implicaciones se pueden ver de manera gráfica en la figura E.2.

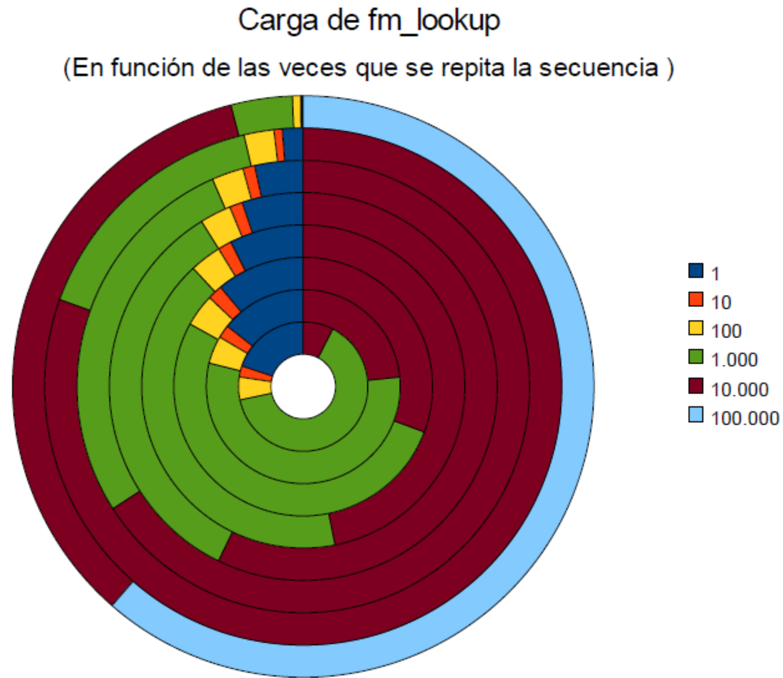

Figura E.2 Carga de la función fm_lookup en función de las veces que se repite la secuencia dada

Esto puede tener repercusiones dramáticas en la ejecución del algoritmo dado que ese 1% de cadenas pueden llegar a formar un cuello de botella para la función fm_lookup. De este modo, conviene tener en cuenta este dato a la hora de diseñar la estrategia de marcado y estructuras auxiliares para la decodificación.

Por último, se analizó con qué frecuencia se dan cadenas homogéneas de bases en el índice FM. Recordar que los aceleradores del índice aprovechaban esta distribución característica para ahorrar operaciones de popCount. Por ello es interesante comprobar en que tamaño se da una mayor probabilidad de acelerar el conteo y salvar la operación crítica del kernel. En la figura E.3 podemos ver los resultados. Se aprecia como la entropía para cubetas de más de 8 caracteres aumenta logarítmicamente mientras que las posibilidades de tener de 1 a 3 aceleradores disminuyen drásticamente. Un tamaño de bloque de 8 es inviable dado que ocuparía demasiado espacio. Es por ello que se concluye que no merece la pena reducir el tamaño de bloque más allá de 32 caracteres.

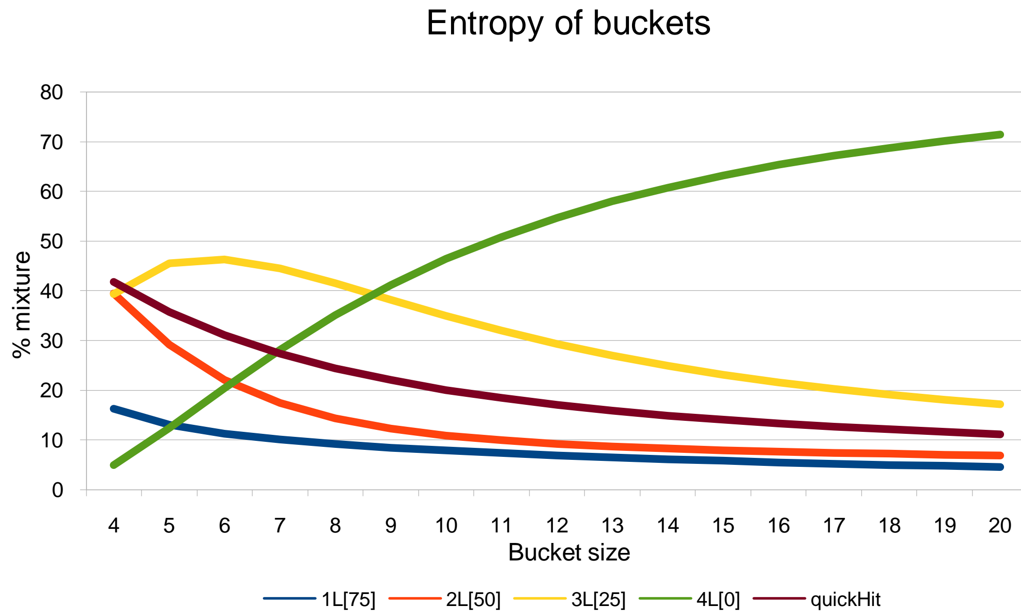

Figura E.3 Probabilidad de encontrar bloques con acelerador y la entropía de las cubetas

F. Aplicaciones para el análisis de la biblioteca y del índice

A continuación se describe brevemente las principales aplicaciones generadas para analizar y evaluar el rendimiento o la estructura de la librería GEM.

GenerateChains

GenerateChains es una aplicación dedicada a extraer cadenas del índice de referencia y generar ficheros de cadenas de pruebas. Posteriormente estos ficheros son utilizados por aplicaciones como searchTest para realizar los test de rendimiento. Esta aplicación permite generar bancos de pruebas con un porcentaje determinado de cadenas contenidas en el índice y el resto generdas aleatoriamente. Ademas, permite incorporar fallos en la codificación de las cadenas en un porcentaje dado para simular los fallos de las maquinas de secuenciación autenticas.

SearchTest. DoTest

SearchTest y el modulo doTest son aplicaciones programadas para evaluar el rendimiento de implementaciones concretas de GEM. Es decir, una vez implementada una alternativa que cumpla con la API de GEM, searchTest ejecuta pruebas sobre ese modulo. Se puede configurar para generar benchmarks para diferentes rangos de cadenas de entrada, tamaños del banco de pruebas, etc. Además permite lanzar mediciones de tiempo de cada parte individualmente o la medición agregada de todas las partes. El programa ha sido implementado con la finalidad de ahorrar tiempo en las evaluaciones de prestaciones procurando reducir la instrumentalización del código y que esta no afecte a las mediciones.

Además permite la generación de ficheros de resultados con la ejecución del programa que pueden ser después comprobados con CheckFile. El objetivo último es minimizar el número de fallos producidos en sucesivas fases de implementación.

SearchStats

SearchStats es una aplicación dedicada a recoger datos de la ejecución de la librería GEM. Esta programada de forma que instrumentaliza la librería para calcular valores como el número de bucles que cumplen la condición hi-lo=1, cuantas iteraciones llegan hasta cierta condición, cuantas iteraciones salen del bucle por un condición dada, etc. Este programa no mide el tiempo de ejecución, por lo que no se presta atención a la elevada instrumentalización del código que genera. Una ejecución normal de SearchStats cuesta 5 veces más que una de SearchTest. No obstante, proporciona datos muy útiles para el análisis cuantitativo de la ejecución del método FM en la librería GEM.

Además, con SearchStats se pueden generar graficas compatibles con Octave que indiquen varios parámetros de la distribución de accesos al índice. Por ejemplo, GAO, GAO-color, etc.

LookupStats

LookupStats es la aplicación análoga a searchStats pero aplicada al procedimiento de fm_lookup. En general, realiza las mismas funciones, pero además permite la generación de datos relacionados con el marcado del CSA, las distancias entre anclajes, etc.

CheckFile

CheckFile se encarga de recoger los ficheros de prueba generados con SearchTest y comprobar que todos los resultados son correctos. Su ejecución es más lenta que la de searchTest ya que debe comprobar los resultados uno a uno. No obstante permite tener un 100% de fiabilidad de que las implementaciones son correctas. Además, permite el cotejo de resultados entre fichero de pruebas. Es decir, una vez comprobado que un fichero es correcto, se pueden cotejar con este ficheros de resultados de diferentes bechmarks que utilizaron los mismos datos de entrada.

DNAstatsGrouping

DNAstatsGrouping es una herramienta dedicada a examinar el índice en busca de datos como la distribución de cadenas en el mismo, las repeticiones, el número de repeticiones, la carga generada por una cadenas, cotejar modelos de distribución con los del índice, etc.

CSAStats

CSAStats es una aplicación cuya finalidad es proporcionar datos sobre diferentes estrategias de marcado del CSA sin necesidad de implementarlas. Permite obtener datos del desempeño de las diversas técnicas para diferentes parámetros rápidamente y sin necesidad de implementarlas. Simplemente simula las técnicas de marcado y muestra los datos obtenidos.

G. Bibliografia

[1]. P. Ferragina and G. Manzini. Opportunistic data structures with applications. In Proc. FOCS'00, pp. 390–398, 2000.

[2]. P. Ferragina and G. Manzini. An experimental study of an opportunistic index. In Proc. SODA'01, pp. 269–278, 2001.

[3]. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.

[4]. Robert M. Gray. Entropy and Information Theory. Springer-Verlag.1990

[5]. P. Ferragina, G. Manzini, V. M¨akinen, and G. Navarro. An alphabet-friendly FMindex. In Proc. SPIRE'04, pp. 150–160, 2004. LNCS 3246.

[6]. Sz. Grabowski, V. M¨akinen, G. Navarro, and A. Salinger. A simple alphabetindependent FM-index. In Proc. PSC'05, pp. 230–244, 2005.

[7]. R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes.In Proc. SODA'03, pp. 841–850, 2003.

[8]. Mäkinen, V. and Navarro, G. 2004c. Run-length FM-index. In Proc. DIMACS Workshop: "The Burrows-Wheeler Transform: Ten Years Later" (Aug. 2004), pp. 17–19.

[10]. G. Jacobson. Succinct Static Data Structures. PhD thesis, CMU-CS-89-112, Carnegie Mellon University, 1989.

[11]. R. Przywarski, Sz. Grabowski, G. Navarro, and A. Salinger. FM-KZ: An even simpler alphabet-independent FM-index. In Proc. PSC'06, 2006.

[12]. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. DEC SRC Research Report 124, 1994.

[13]. http://valgrind.org/info/tools.html

[14]. James Reinders. VTune Performance Analyzer Essentials