Aceleración de Algoritmos de Emparejamiento de Secuencias Genéticas

Gutiérrez Martín, Francisco.

Resumen— El emparejamiento de secuencias genéticas se puede definir como la búsqueda de las diferencias que existen entre dos cadenas de caracteres: patrón y texto. Existe una gran variedad de algoritmos que resuelven este problema con diferentes costes computacionales. En este trabajo se consideran algoritmos de emparejamiento de tipo exacto y, utilizando técnicas de la Ingeniería de Rendimiento, se mejora la eficiencia de codificación de los mismos. Se obtiene un *Speed Up* de 3,7x para un algoritmo basado en el peor caso y un *Speed Up* de 8x en un algoritmo basado en casos medios. Como resultado final se obtiene una versión del programa con un *Speed Up de* 114x frente a la versión base, para un patrón de 10k caracteres y un texto con el 20% de error respecto al patrón.

Palabras clave— Distancia de Edición, Programación Dinámica, Alineamiento Diagonal, Matriz de Distancia, Ingeniería de Rendimiento, Computación de Altas Prestaciones, Bioinformática.

Abstract— The matching of genetic sequences can be defined as the search for the differences between two-character strings: pattern and text. A great variety of algorithms can solve this problem with different computational costs. This work studies two exact string matching algorithms and, using Performance Engineering techniques, improves its coding efficiency, attaining a Speed Up of 3,7x for a worst case based algorithm and a Speed Up of 8x on an average case algorithm. As a final result a version with an Speed Up of 114x is achieved compared to the base version, for a 10k character pattern and a text with a 20% difference compared to the pattern.

Index Terms— Edit Distance, Dynamic Programming, Diagonal Matching, Distance Matrix, Performance Engineering, High Performance Computing, Bioinformatics.

1 Introducción

Las secuencias de ADN y de proteínas se codifican como textos largos utilizando alfabetos específicos ({A, C, G, T}, en secuencias de ADN). Estas secuencias representan el código genético de los seres vivos. La búsqueda de secuencias específicas sobre estos textos es fundamental para resolver problemas tales como el ensamblaje de cadenas de ADN a partir de sub-cadenas obtenidas por experimentos, hallar características particulares en distintas cadenas o determinar cuán diferentes son entre sí para reconstruir los árboles evolutivos.

En un principio todo esto fue enfocado como la búsqueda de un patrón dado en un texto. Sin embargo, la búsqueda exacta es de poca utilidad para este tipo de aplicaciones, ya que los patrones raramente coinciden exactamente con el texto; las medidas experimentales tienen errores de diferentes tipos e incluso las cadenas correctas pueden tener pequeñas variaciones, algunas de ellas importantes debido a que representan mutaciones y alteraciones de carácter evolutivo.

Todos los problemas relacionados con el alineamiento entre secuencias requieren de un concepto de similitud, así como un algoritmo para calcularlo. Esto dio una motivación para realizar búsquedas permitiendo errores, siendo los errores aquellas operaciones que los biólogos saben que es común que ocurran en secuencias genéticas. La distancia entre dos secuencias se definió como la secuencia de operaciones mínima para transformar una cadena en otra. A dichas operaciones se les asignara un coste, de modo que las operaciones más comunes sean las más baratas. El objetivo es entonces minimizar el coste total.

La definición formal del problema es:

Sea $T \in \Sigma^+$ un texto de longitud $n=|T|$
Sea $P \in \Sigma^+$ un patrón de longitud $m=|P|$
Sea $k \in \mathbb{R}$ el máximo error permitido.
Sea d: $\Sigma^+ \times \Sigma^+ \rightarrow \mathbb{R}$ una función de distancia
Dados T, P, k y d( ), devolver el conjunto de todas las posiciones de texto j tales que exista i y tal que $d(P, Ti...j) \leq k$

Restringiéndonos ahora a un sub-conjunto de posibles funciones de distancia, consideremos sólo a aquellas definidas de la siguiente forma:

La distancia $d(x, y)$ entre dos cadenas x e y, es el coste mínimo de una secuencia de operaciones que transforman x en y. El coste de una secuencia de operaciones es la suma de los costes de una operación individual. Las operaciones son un conjunto finito de reglas con forma $\delta(z, w) = c$, donde z, w son diferentes cadenas y c es un número real. Una vez se ha convertido la sub-cadena z en w, no se pueden realizar más operaciones en w.

En cuanto a las operaciones, en la mayoría de aplicaciones quedan restringidas a:

- Inserción: $\delta(\varepsilon, a)$, p. ej. inserción de la letra a.
- Deleción: $\delta(a, \varepsilon)$, p. ej. deleción de la letra a.
- Sustitución o Reemplazo: $\delta(a, b)$ para $a \neq b$, p. ej. sustitución de a por b.

Llegados a este punto, se pueden definir las funciones de distancia más comúnmente utilizadas:

- Distancia de Hamming [2]: se permiten únicamente sustituciones, de coste 1 en su definición simplificada. En la literatura es en muchos casos llamado emparejamiento de cadenas con k divergencias.
- Distancia de Levenshtein [3]: permite inserciones, deleciones y sustituciones. En su definición simplificada, todas las operaciones cuestan 1. Dicho de otro modo, el mínimo número de inserciones, deleciones y sustituciones para obtener dos cadenas iguales.
- Distancia de Subsecuencia Común más Larga [4, 5]: permite únicamente inserciones y deleciones, ambas de coste 1. El nombre de esta distancia se basa en el hecho de que esta mide la longitud del emparejamiento de caracteres más largo que puede ser realizado entre dos cadenas, respetando el orden de las letras.

El estudio del emparejamiento por aproximación de cadenas es un campo extenso y antiguo. Por ello existe un gran número de métodos en la literatura [1]. Estos métodos pueden ser clasificados en tres grandes grupos:

- Basados en autómatas: Esta área es bastante antigua. Es interesante ya que con este método se consiguen los mejores tiempos en el peor de los casos (O(n)). Sin embargo, tienen una dependencia exponencial en cuanto a tiempo y tamaño en m y k que los limita en cuanto a la práctica [1].
- De filtrado o deterministas: Esta categoría está formada por algoritmos que filtran el texto, descartando rápidamente áreas que no coinciden. Estos tipos de algoritmos se enfocan en el caso medio y son especialmente interesantes al combinarlos con otros que no realizan la búsqueda sobre todos los caracteres del texto (llamados algoritmos de búsqueda local). El mejor logro hasta la fecha es un algoritmo con un coste temporal medio de $O(n(k + \log m)/m)$. En la práctica, los algoritmos de filtrado son los más rápidos. Sin embargo, están limitados en su aplicabilidad por un nivel de error k. Además, dependen de un algoritmo que no sea de filtrado para procesar las potenciales coincidencias [1].
- Basados en programación dinámica: Esta categoría destaca principalmente por los logros en cuanto a complejidad temporal en el peor de los casos ($O(kn)$). En cuanto a los casos aproximados se ha logrado una complejidad $O(kn/\sqrt{o})$. Este tipo de algoritmos son competitivos en la práctica cuando se busca un alineamiento exacto. Sin embargo, también son utilizados conjuntamente a algoritmos deterministas cuando se desea un alineamiento aproximado (se realiza un filtrado mediante algoritmos heurísticos, para procesar con más detalle las posibles coincidencias mediante un algoritmo basado en programación dinámica).

Existen numerosos algoritmos con importantes mejoras en cuanto a su complejidad teórica pero muy lentos en la práctica. Estos algoritmos pueden ser viables en algunos escenarios, pero desafortunadamente estos no son los escenarios comunes que se encuentran en aplicaciones de alineamiento de secuencias genéticas. Por lo tanto, en este trabajo sólo se considerarán los algoritmos o métodos que han demostrado ser prácticos [1].

2 Objetivos

El principal objetivo del presente trabajo consiste en acelerar, mediante técnicas de ingeniería de rendimiento, algoritmos de emparejamiento de secuencias genéticas, aplicando diferentes estrategias de paralelización y mejorando la eficiencia de la codificación de estos.

Para cumplirlo, se planifican varios hitos o sub-objetivos a lograr progresivamente:

1. Estudio y elección de los algoritmos a acelerar.
2. Implementación de las versiones base de los algoritmos.
3. Elección del conjunto de datos con el que trabajarán los algoritmos.
4. Comprobación del correcto funcionamiento de las versiones base.
5. Perfilado de las versiones base, extracción de métricas de rendimiento y análisis del código generado por el compilador.
6. Estudio y aplicación de posibles estrategias o técnicas de ingeniería de rendimiento sobre las versiones base, con el fin de obtener una mejor eficiencia de codificación.
7. Implementación de versiones aceleradas.
8. Comprobación del correcto funcionamiento de las versiones aceleradas.
9. Creación de una biblioteca de lenguaje C debidamente documentada, dónde se recoge de una forma ordenada, todos los avances conseguidos.

3 Metodología

En este apartado se expone la metodología empleada en la aceleración de los algoritmos, comenzando por el criterio de elección de los mismos, así como cada uno de los pasos que se han seguido para llevar a cabo su aceleración.

Toda implementación de código mencionada en este apartado puede ser consultada en el Anexo I, dónde se encuentra un enlace a la biblioteca de este trabajo, de libre uso y acceso bajo licencia GPLV3.

Destacar que todas las pruebas de rendimiento realizadas han sido focalizadas en obtener las medidas única y exclusivamente del cómputo del algoritmo, excluyendo todo cómputo relacionado con la adquisición de datos y asignación de memoria.

3.1 Estudio y elección de los algoritmos a ser acelerados

Visto el gran número de algoritmos que existen para resolver el problema, cabe discutir qué rama de la taxonomía o

algoritmo escoger para estudiarlo en profundidad y realizar su versión acelerada.

En la práctica, los biólogos utilizan métodos de filtrado cuándo se desea obtener un resultado rápido en un tamaño de datos considerable (p. ej. comparación de una determinada cadena contra miles almacenadas en bases de datos) y algoritmos de programación dinámica cuando se requiere realizar una comparación exacta entre un grupo más reducido de cadenas.

Los algoritmos basados en filtros o heurísticos han demostrado ser muy rápidos en la práctica, a expensas de proporcionar un resultado que no es el más óptimo. Por otra parte, los algoritmos basados en programación dinámica son considerablemente más lentos en comparación, pero garantizan, en su mayoría, proporcionar un emparejamiento óptimo.

Se ha decidido escoger algoritmos basados en programación dinámica dado su uso extensivo junto a otros métodos heurísticos, que por lo general utilizan, una vez realizado el filtrado, algoritmos de programación dinámica para procesar las potenciales coincidencias. Por lo tanto, al acelerar un algoritmo basado en programación dinámica, indirectamente se aceleran algoritmos basados en métodos heurísticos.

Centrándose ahora en la categoría que engloba los algoritmos de programación dinámica, queda discutir qué algoritmo considerar. Los algoritmos de alineación de secuencias basados en el peor caso comparan cada uno de los caracteres en una secuencia para cada uno de los caracteres en la otra, un proceso que requiere un tiempo de cálculo proporcional al producto de las longitudes de las dos secuencias. Por otra parte, los algoritmos basados en el caso medio explotan el hecho de que las similitudes de secuencia significativas raramente se encuentran al alinear el final de una secuencia con el comienzo del otra y, por lo tanto, es posible calcular una alineación óptima entre dos secuencias considerando solo esos residuos que se pueden alinear dentro de una banda diagonal con una anchura determinada.

Aunque parece claro que la elección de un algoritmo basado en el caso medio dará un mejor tiempo de partida en la versión inicial, finalmente se ha decidido escoger un algoritmo perteneciente a cada una de las sub-ramas, con el objetivo analizar cómo responde cada uno de ellos ante distintas estrategias de aceleración y poder realizar una comparación empírica entre ellos, confirmando así, mediante datos reales, la menor complejidad temporal de los algoritmos basados en casos medios frente a los algoritmos basados en el peor caso.

Los algoritmos escogidos son:

- Matriz de Distancia, como algoritmo exacto.
- Wavefront Aligner, como algoritmo basado en casos medios [6].

3.2 Elección del conjunto de datos

Con el fin de obtener resultados de tiempo realistas en las diversas pruebas empíricas a las que se han sometido las diferentes versiones de los algoritmos, es necesario un conjunto de datos que se ajuste a la realidad. Para ello, se ha implementado un generador de secuencias.

El programa permite crear aleatoriamente un patrón y a partir de éste, crear un texto a partir de un porcentaje de error configurable. El tipo de error que se introduce en el texto también es configurable, pudiendo elegir entre inserciones, deleciones o sustituciones.

El tamaño del problema ha sido fijado a un patrón de 10k caracteres y un texto generado a partir de este, con un porcentaje de error del 20%. El tipo de errores introducidos en el patrón han sido aleatorios y uniformemente distribuidos.

3.3 Aceleración del algoritmo Matriz de Distancia

3.3.1 Análisis computacional

Observando la definición del Algoritmo 1, se aprecian dos etapas diferenciadas:

- Asignación e inicialización de las estructuras de datos necesarias para el cómputo (líneas 3 a 7).
- Procesamiento de la matriz (líneas 8 y 9).

```
Sea n el tamaño del patrón
Sea m el tamaño del texto.
1: Si n = 0, devolver m.
2: Si m = 0, devolver n.
3: Construir una matriz con n + 1 filas y m + 1 columnas.
4: Inicializar la primera fila de la matriz con la secuencia 0,
1, 2, ..., m; y la primera columna de la matriz con la secuencia 0, 1, 2, ..., n.
5: Colocar cada carácter de la cadena A en su correspondiente
celda i (i va de 1 a n).
6: Si A(i) = B(j) el costo de la celda (i,j) es 0.
7: Si A(i) != B(j) el costo de la celda es 1.
8: Para cada celda de la matriz, su valor es el mínimo de :
   Valor de la celda (i-1,j) + 1 (Deleción)
   Valor de la celda (i,j-1) + 1 (Inserción)
   Valor de la celda (i-1,j-1) + costo de celda (Sustitución)

9: La distancia de edición es el valor de la celda (n,m).
```

Algoritmo 1. Matriz de distancia de edición.

En la primera etapa, se deduce la complejidad espacial del problema. Se necesita un espacio que permita alojar los dos vectores que representan las cadenas, de tamaño n y m respectivamente, y la matriz donde se realizarán los cálculos con un tamaño de $(n+1)*(m+1)$. El programa ocupará en memoria entonces $((n+1)*(m+1)) + n + m$. De la segunda etapa se puede deducir que será donde el computador realizará el trabajo más significativo. En primer lugar, se rellena la matriz a partir de los resultados de las comparaciones entre los elementos de las dos cadenas; esta operación se realiza fijando un elemento de la primera cadena y comparándolo con cada uno de los elementos de la segunda

cadena, para cada fila de la matriz. El computador, entonces, tendrá que realizar *n*m* comparaciones. Este fragmento del algoritmo es altamente susceptible a ser acelerado, ya que cada una de las comparaciones se pueden realizar independientemente.

Una vez el computador ha realizado todas las comparaciones pertinentes, quedará por realizar la sección dónde para cada celda, se calcula el mínimo entre *((i-1,j) + 1, (i,j-1) + 1, (i-1,j-1)*. Este patrón de cómputo se puede identificar como una reducción recurrente 3:1, ya que el valor que toma una celda viene dado de una reducción de celdas adyacentes a ella. Por lo tanto, la complejidad computacional de este fragmento del problema es de *n*m* operaciones de reducción 3:1.

En síntesis, y tras analizar el problema desde una perspectiva computacional, el problema conlleva un tamaño con las siguientes complejidades:

- Complejidad temporal: O(m*n).
- Complejidad espacial: O(m*n).

3.3.2 Perfilado de la versión base.

Se realiza un primer perfilado de la ejecución del programa objeto. Se extraen las métricas proporcionadas por el profiler y con ellas se elaboran los primeros datos de rendimiento.

Se obtiene IPC de 1,8 sobre los 4 que el procesador puede ofrecer como máximo.

En cuanto al uso del ancho de banda de la memoria principal, se emplea un máximo del 40% de ésta, por lo que el programa no está limitado en este ámbito.

Analizando el código ensamblador producido por el compilador, se observa cómo éste ha sido capaz de vectorizar con instrucciones SSE tanto la función que procesa la matriz como en la que la inicializa.

La función que más tiempo lleva en la ejecución es la encargada de realizar el procesado de la matriz (aproximadamente un 60% del tiempo total), por lo que las primeras optimizaciones del programa se centran en dicha función.

3.3.3 Optimizaciones.

3.2.3.1 *Diagonal Strip Traverse*

El principio de esta mejora es cambiar el patrón con el que se recorre la matriz para evitar las dependencias entre iteraciones del bucle interno.

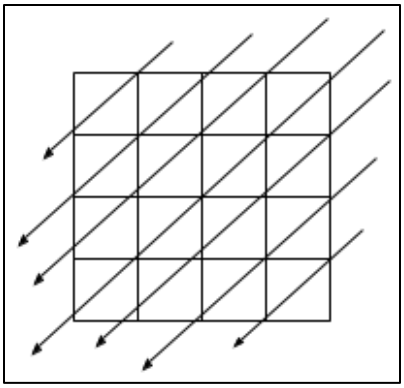

Fig 1. Accesos *Diagonal Strip Traverse* a los elementos de la matriz.

Aprovechando que las dependencias se encuentran en las celdas superior-izquierda, se puede recorrer la matriz en tiras anti-diagonales, tal y como se muestra en la Figura 1, de manera que cada elemento de la diagonal pueda ser computado de manera independiente por cada núcleo del procesador.

Aunque con este cambio se explota el paralelismo en el procesamiento de las celdas, la forma en la que se accede en éstas no es óptima. Para cada operación, se tiene que acceder a un *stride*-m, *stride*-1, y *stride*-m-1 respectivamente a la posición de la celda a calcular, esto causa un mal uso de la localidad espacial y temporal de la caché, desembocando en un peor tiempo de ejecución respecto a la versión base.

3.3.3.2 *Diagonal Strip-Tiling Traverse*

El objetivo principal de esta mejora es conservar el patrón de acceso a memoria que hace posible el paralelismo del cómputo de la matriz, pero minimizando el mal uso de la localidad espacial que dicho patrón de acceso a memoria realiza.

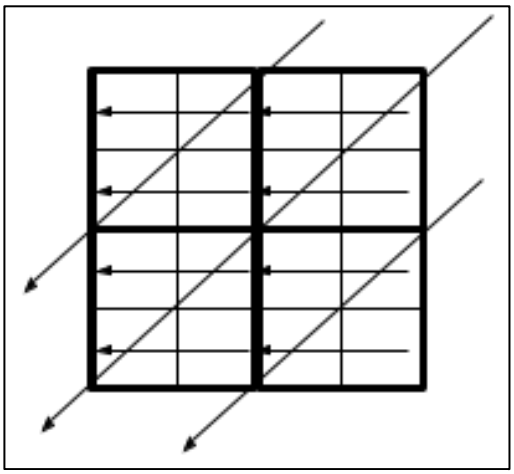

Fig. 2. Accesos "Diagonal Strip-Tiling Traverse" a los elementos de la matriz.

Para ello, en lugar de recorrer cada elemento de la diagonal individualmente, se agrupan dichos elementos en "tiles" o subgrupos, de manera que cada subgrupo puede ser recorrido con un acceso a memoria más uniforme, tal y como se muestra en la Figura 2.

Esto permite aprovechar tanto la localidad espacial (cuando se carga un elemento, para las siguientes iteraciones los próximos ya estarán cargados en cache, para cada bloque) como la localidad temporal (en el cómputo de la siguiente fila, la anterior ya estará previamente cargada). El tamaño de cada subgrupo viene determinado por la variable *t*, que ha sido calculado de tal forma que cada subgrupo quepa en la caché privada L2 de cada núcleo del procesador utilizado. Teóricamente, el número de elementos que tiene que contener cada subgrupo, para que quepa en la caché privada, es de $(n/t)2 * s = c$, siendo n el tamaño de los lados de una matriz cuadrada, s los bytes que ocupa

cada elemento contenido en ella y *c* el tamaño en bytes de la caché privada.

3.4 Aceleración del algoritmo Wavefront Aligner

3.4.1 Análisis computacional

Se trata de un algoritmo categorizado como *diagonal stroke* ya que está ideado para mejorar el caso medio realizando una alineación por banda diagonal. Este algoritmo explota las similaridades entre texto y patrón para calcular los alineamientos posibles de menor a mayor error. Cabe destacar que el algoritmo no precisa alojar en memoria la matriz utilizada en otras soluciones basadas en programación dinámica. En su lugar, este algoritmo se sirve de una estructura *strokes* (Figura 3), dónde en cada elemento o casilla se guardan los offsets diagonales o, dicho de otro modo, el número de caracteres del texto y del patrón que, respectivamente para cada diagonal, son coincidentes [6].

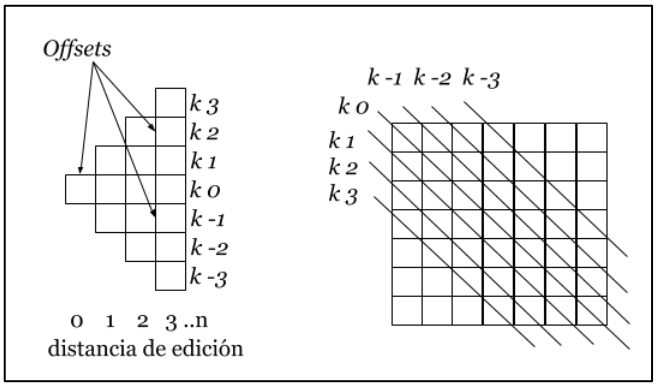

Fig. 3. Estructura de datos utilizada por el algoritmo Wavefront Aligner (izquierda) y su representación en una matriz de programación dinámica (derecha).

Observando el Algoritmo 2 se pueden diferenciar dos partes:

- Extender: Cálculo de los offsets de cada diagonal o, dicho de otra forma, cálculo de cuántos elementos del patrón y del texto son coincidentes desde la posición de cada diagonal (líneas 4 a 10).
- Ampliación: Se incrementa en dos el número de diagonales efectivas y se inicializan para poder ser computadas en la siguiente iteración (líneas 13 a 18)

El algoritmo comienza computando primera diagonal *k0* y no extiende su anchura hasta que ésta se encuentra con un par de caracteres no coincidentes. Cuando esto ocurre, se puede decir que la distancia de edición se incrementará una unidad y acto seguido se extiende la banda con dos diagonales adicionales. Este proceso se repite para cada una de las diagonales hasta que la k-ésima diagonal ecuador (la diagonal que, en una matriz de programación dinámica, cruza desde la casilla superior izquierda hasta la casilla inferior derecha) llega al final de su recorrido. Una vez acabado todo el proceso, la distancia de edición corresponde al número de veces que la banda diagonal haya sido ampliada.

La complejidad del algoritmo depende, por tanto, de la distancia de edición máxima que haya entre el patrón y el texto, siendo su complejidad espacial $O(k^2)$ y su complejidad temporal $O(k*l)$, siendo k la máxima distancia de edición entre patrón y texto y l la longitud mínima entre patrón y texto.

```
Sea n la longitud del patrón
Sea m la longitud del texto
Sea strokes una estructura de datos como la que se
muestra en la figura 2.
1: Inicializar d a 0
2: Inicializar strokes[0][0] a 0
3: Mientras d < m+d
4:      Para k =-d a k=d
5:          Calcular la posición del carácter cn
            del patrón en función de strokes[d][k]
            (offset de la diagonal k)
6:          Calcular la posición del carácter cm
            del texto en función de strokes[d][k]
            (offset de la diagonal k)
7:          Mientras cn==cm
8:              Incrementar stokes[d][k]
9:              cn = cn+1
10:             cm = cm+1
11:     Si abs(m-n) <= d y strokes[d][m-n]==t
12:         La distancia de edición es d, acabar
            ejecución.
13:     strokes[d+1][-d-1] = strokes[-d]
14:     Para k =-d a k=d
15:         strokes[d+1][k] = Max(strokes[d][k],
16:         (strokes[d][k-1]+1), strokes[d][k+1])
17:     strokes[d+1][d+1] = strokes[d]
18:     Incrementar d
```

Algoritmo 2. Wavefront Aligner.

3.4.2 Perfilado de la versión base

Se obtiene un IPC de 1,4 y un uso del ancho de banda de la memoria principal del 37%. Se observa que el compilador logra vectorizar la función encargada de ampliar las diagonales, utilizando instrucciones AVX. La sección crítica del programa se sitúa en la función que extiende o computa los *offsets* para cada diagonal, llevándose un 70% del tiempo de ejecución del algoritmo. Por tanto, se decide centrar las optimizaciones del programa en esta última función.

3.4.3 Optimizaciones

3.4.3.1 *Packed Extend*

El objetivo de esta primera optimización es aplicar una paralelización SIMD en la función encargada de extender las diagonales. Esta mejora, representada en la Figura 4, se basa en empaquetar ocho caracteres del texto y del patrón para realizar una comparación vectorial. Si todos los caracteres del paquete son iguales, se empaquetan los siguientes

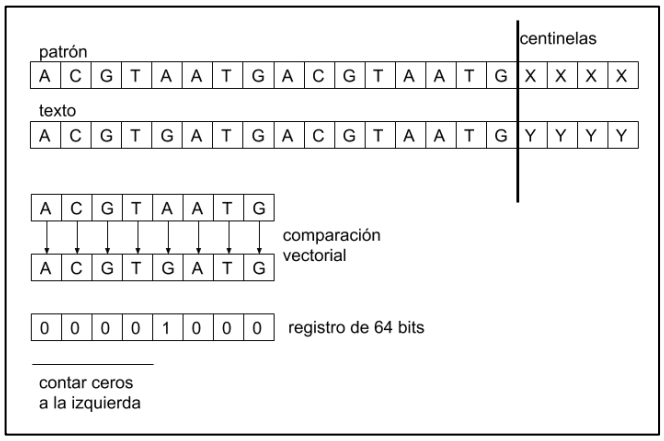

Fig. 4. Principio de la mejora *Packed Extend*.

ocho y se realiza de nuevo la comparación. Este proceso se repite hasta que se encuentra que todos los elementos de la comparación vectorial no son iguales. En este caso, se cuenta el número de elementos que son iguales, empezando desde la izquierda hasta el primer elemento no igual de la derecha. Para evitar instrucciones de comparación adicionales destinadas a marcar el final de las cadenas, se emplean unos centinelas al final de cada cadena de caracteres a modo de condición final. Dicho de otro modo, son unos caracteres que aseguran la no igualdad en el caso de realizar una comparación entre ellos y cualquier carácter del conjunto que conforma el texto y patrón.

Con la aplicación de esta mejora se aumenta significativamente el IPC gracias a la reducción de saltos condicionales, los cuales son costosos en ciclos debido a los fallos de predicción.

3.4.3.2 *Packed Extend MIMD grano fino*

Esta mejora se basa en aprovechar la no dependencia que existe entre el cálculo de *offsets* de cada diagonal ya que, para cada iteración, cada una de las diagonales existentes computa su offset de manera independiente, tal y como se muestra en la figura 5. La implementación está paralelización se ha realizado utilizando *OpenMP*.

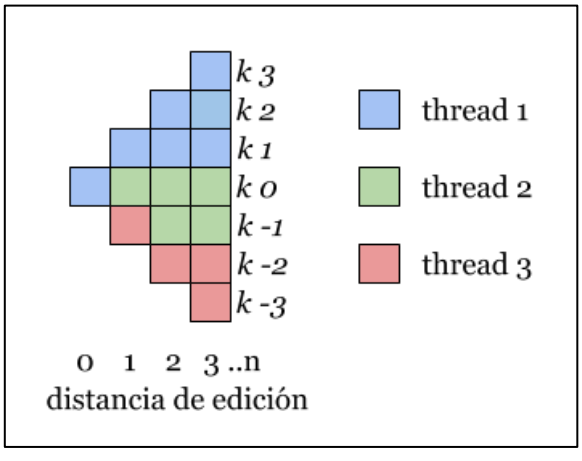

Fig. 5. Principio de paralelización MIMD de la función *extend*.

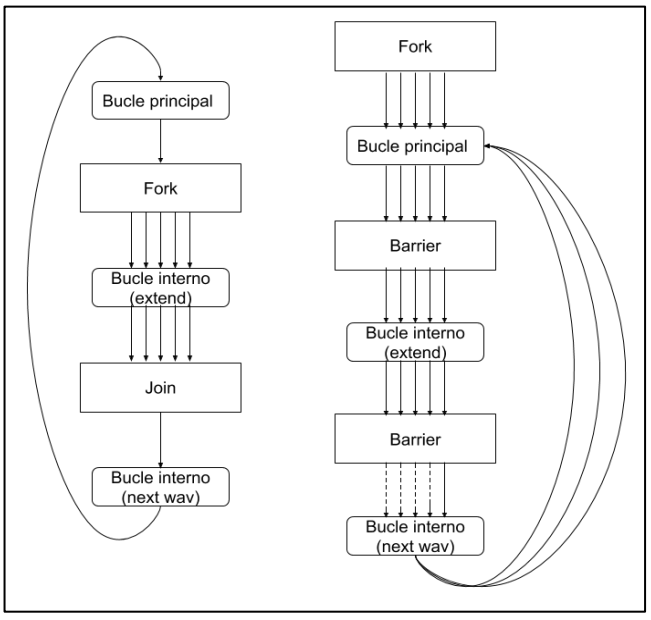

Fig. 6. Traza de creación y destrucción de threads para la versión *packed extend MIMD* (izquierda) y la versión *packed extend MIMD V2* (derecha).

3.4.3.3 *Packed Extend MIMD V2*

La idea de esta mejora es evitar en la medida de lo posible todo *overhead* relacionado con la creación de *threads*. En la Figura 6 se muestra la comparación de la traza de ejecución para la anterior versión y la actual. Como se puede observar, en el caso de la anterior versión, se crean los *threads* tantas veces como iteraciones haya en el bucle externo. La idea es crear todos los *threads* fuera del bucle externo y gestionarlos dentro de éste.

3.4.3.4 *Packed Extend MIMD grano grueso*

En la práctica, cuando se desea realizar un emparejamiento de secuencias, se computa un patrón contra múltiples textos. Tiene sentido, entonces, implementar una versión paralela de forma que cada hilo de ejecución compute los distintos emparejamientos de secuencias. La implementación se ha realizado siguiendo el principio que se muestra en la Figura 7, utilizando *OpenMP*.

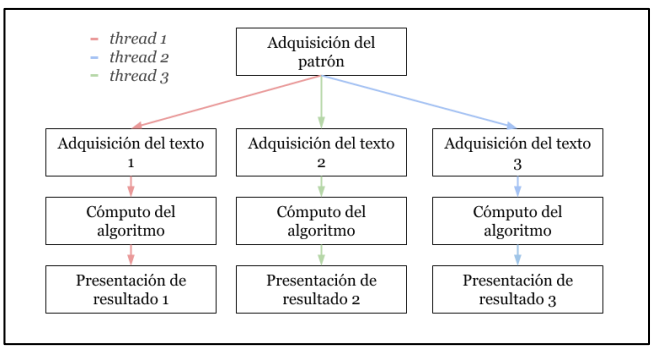

Fig. 7. Principio de paralelización MIMD de grano grueso.

4 Resultados

4.1 Aceleración del algoritmo DP- Matrix

En la Figura 8 se resumen las mejoras obtenidas en la aceleración del algoritmo. El máximo *Speed Up* obtenido es de un 3,7 para la versión paralela *Diagonal tiling traverse* ejecutada en 4 *threads*.

Se aprecia como la versión *Diagonal tiling traverse* responde positivamente ante una paralelización *MIMD*. La eficiencia para la ejecución en 2 y 4 *threads* es muy parecida y cercana al 100% en ambos casos, lo que indica una buena escalabilidad del programa ante el aumento de recursos.

Las ejecuciones en 2 y 4 *threads* de la versión *Diagonal traverse* no han mostrado un aumento en el rendimiento respecto a la versión base. Esto es debido a su limitación por latencia de la memoria principal, causada por el mal aprovechamiento de la localidad temporal y espacial de la caché.

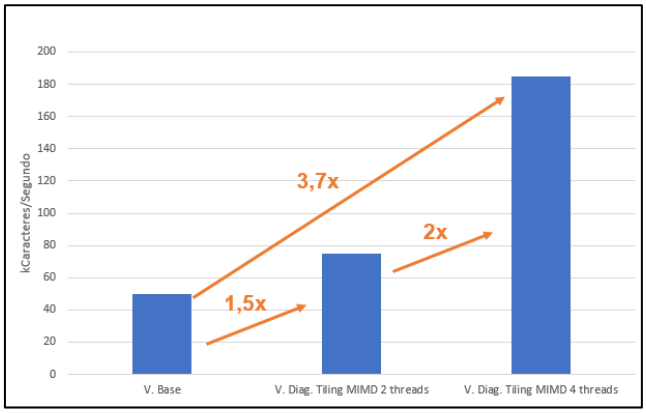

Fig. 8. Resultados de la aceleración del algoritmo *DP-Matrix*

4.2 Aceleración del algoritmo Wavefront Aligner

En la Figura 9 se muestran las mejoras obtenidas de la versión *Packed Extend* respecto a la versión base. Se observa que el principal motivo del aumento en el rendimiento del programa es el aumento del *throughput* de la arquitectura; el cambio respecto a la versión base ha reducido el número de saltos condicionales ejecutados, los cuales tienen un mayor coste de ciclos debido a los fallos de predicción.

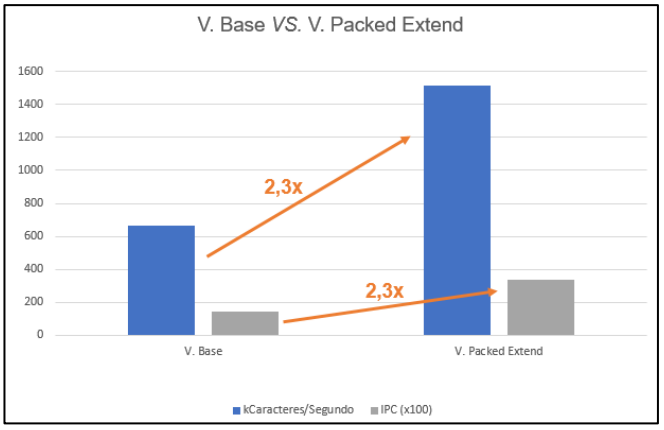

Fig. 9. Versión base *vs.* Versión *packed extend.*

Observando el comportamiento de la versión *Packed Extend* ante una paralelización MIMD de grano fino, en la figura 10, se aprecia escalabilidad hasta aumentar los recursos en 3. A partir de dicha cifra no existe aumento en el rendimiento. Esto es debido a la carga irregular de trabajo que existe en el bucle interno. Por lo general algunas diagonales se extienden considerablemente mientras que la gran mayoría lo hacen muy poco. Se ha implementado una versión con un *scheduler* dinámico con el fin de repartir mejor la carga entre los diferentes recursos, pero la sobrecarga que genera dicho *scheduler* es superior al tiempo medio que ocupa procesar la extensión de una diagonal, por lo que su uso no ha resultado en una mejora de rendimiento.

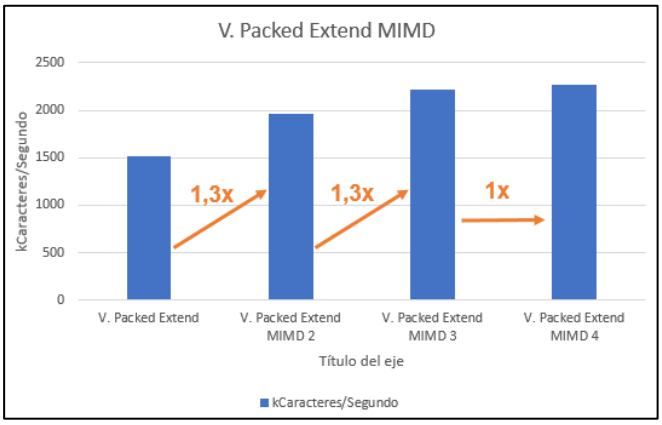

Fig. 10. Comportamiento de la versión *packed extend* ante una paralelización de grano fino.

4.3 Comportamiento de los algoritmos frente a diferentes tasas de error

En la figura 11 se muestra es comportamiento de la mejor versión del algoritmo DP-Matrix frente a la versión base del algoritmo Wavefront Aligner. Aunque en tasas de error superiores al 40% el algoritmo DP-Matrix ofrece un mejor rendimiento, esto no implica ningún avance ya que los emparejamientos de secuencias empiezan a ser relevantes a partir del 20% de error, tasas donde el algoritmo Wavefront Aligner se muestra claramente superior.

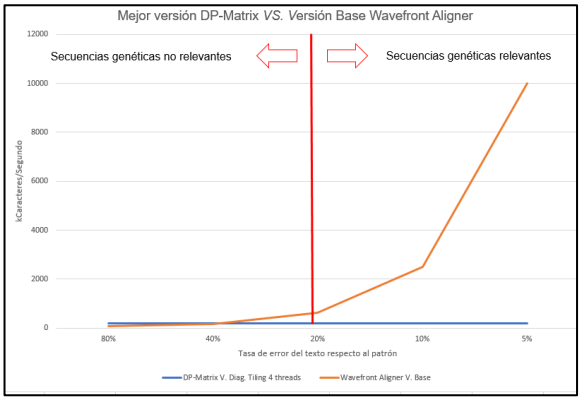

Fig. 11. Comportamiento de los algoritmos frente a diferentes tasas de error.

4.4. Paralelización de grano grueso del algoritmo Wavefront Aligner

Con esta estrategia de paralelización se obtiene una mejor eficiencia, como se muestra en la Figura 12. La eficiencia para 2 y 3 *threads* es muy cercana al 100% mientras que para 4 *threads* esta cifra se ve reducida al 85%. Esto es debido a que el ancho de banda de la memoria principal, para 4 *threads*, se muestra saturada con una media de uso del 90%.

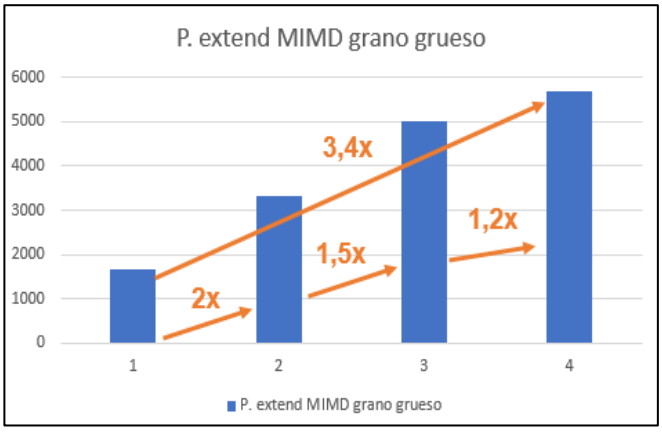

Fig. 12. Comportamiento de la versión *packed extend* ante una paralelización de grano grueso.

5 Conclusiones

Los avances logrados se resumen en la Figura 13. Se ha obtenido desde la primera versión a la última un *Speed Up* de 114. Este resultado se ha logrado en gran medida gracias a la eficiencia del algoritmo *Wavefront Aligner*, el cual ha sido acelerado 8,5 veces utilizando diversas estrategias de paralelización e ingeniería de rendimiento.

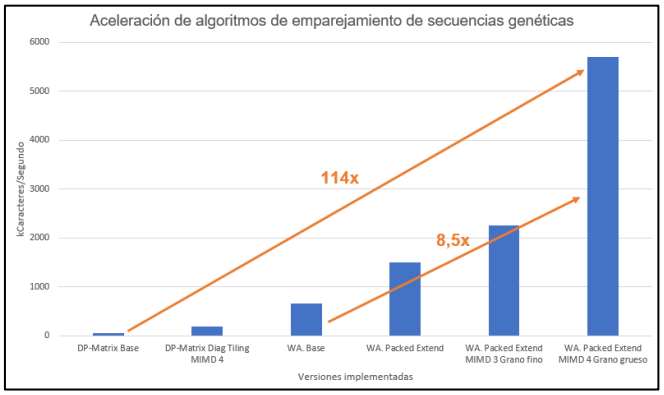

Fig. 13. Resumen de los resultados obtenidos.

El mejor resultado ha sido logrado utilizando una paralelización de grano grueso. La limitación en este caso viene dada por el ancho de banda de la memoria. Teniendo en cuenta que la paralelización ha sido aplicada en una arquitectura UMA, la barrera que el ancho de banda de la memoria supone se puede romper aplicando la misma estrategia en una arquitectura NUMA donde no se comparta el bus de datos o en una aceleradora, donde el ancho de banda es mayor.

La paralelización en grano fino sin embargo no es trivial. La diferencia en cuanto a carga de trabajo que existe en el bucle interno hace difícil esta tarea, por lo que de seguir avanzando en la aceleración del algoritmo habría que apostar por una estrategia de grano grueso.

Agradecimientos

A mi madre Catalina, por enseñarme desde bien pequeño que el saber no ocupa lugar. A mi padre Francisco, por su apoyo moral y económico. A mis hermanas Encarni y Mari, por su apoyo incondicional. A mi amigo Manuel, por todos los conocimientos que hemos adquirido juntos a lo largo de la carrera. A mis profesores de la mención en Ingeniería de Computadores, en especial a Juan Carlos, Santiago, Eduardo, Ana y Dolores, por su excelente labor docente. A mis tutores de TFG, por su paciencia y buenos consejos. Sin todas las personas mencionadas el presente trabajo no podría haberse llevado a cabo.

Referencias

[1] Gonzalo Navarro. A Guided Tour to Approximate String Matching. Dept. of Computer Science, Uni versity of Chile. 2001.

[2] D. Sanko and J. Kruskal, editors. Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison. Addison-Wesley, 1983.

[3] V. Levenshtein. Binary codes capable of correct ing spurious insertions and deletions of ones. Problems of Information Transmission, 1:8-17, 1965.

[4] Apostolico and C. Guerra. The Longest Common Subsequence problem revisited. Algorithmica, 2:315{336, 1987

[5] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. Journal of Molecular Biology, 48:444{453, 1970.

[6] E. W. Myers. A O(ND) Difference Algorithm and Its Variations. 1986.

ANEXO I: BIBLIOTECA C DEL PROYECTO

https://drive.google.com/drive/folders/1oUYPy-iR-SskFz-BUW4mpri0Cpd-tqIY?usp=sharing

ANEXO II: CARACTERÍSTICAS DEL ENTORNO

CPU[1]	
Arquitectura	Haswell (x86-Intel64)
Frecuencia	4 GHz
Cores/Threads	4 cores x 2 threads por core
Caché L3	8MB, Compartida, Asociativa
Caché L2	4 x 256KB, Privada, Asociativa
Caché L1	4 x 32KB, Privada, Asociativa
Extensiones	MMX, SSE, SSE2, SSE3, SSSE3, SSE4, AES, AVX, AVX2, BMI, F16C

Tabla 1: Características de la CPU utilizada.

Memoria Principal	
Capacidad	16 GB
Frecuencia	2400MHz
Velocidad de transferencia	19 GB/s[2]

Tabla 2: Características de la memoria principal utilizada.

Sistema Operativo	Ubuntu 16.04
Profiler	Intel VTune 2018
Compilador	GCC 7.2[3]

Tabla 3: Sistema operativo, compilador y software utilizado.

[1] La función Turbo Boost de la CPU se ha deshabilitado para las pruebas.
[2] Ancho de banda medido empíricamente.
[3] Todas las compilaciones se han realizado utilizando el flag -O3.