Análisis de BLAST en multicores.

Pedro Erencia, Alejandro Chacón, Juan Carlos Moure, Antonio Espinosa, Porfidio Hernández[1]

Resumen— **Este artículo presenta un análisis del comportamiento de BLAST, concretamente de su aplicación para el alineamiento de proteínas *blastp*, en entornos multicore. Se realizan pruebas con secuencias de diferente tamaño y diferente calidad contra la misma base datos para establecer principios de utilización de recursos y escalabilidad. Se realiza un profiling a nivel de las diferentes etapas del algoritmo para establecer cuales consumen el mayor tiempo y en cuales la calidad y la longitud de la secuencia tienen un mayor impacto.**

Palabras clave— **BLAST, multicores, blastp, alineamiento de secuencias**

I. Introducción

El alineamiento de secuencias es un modelo para la extracción de similitud entre secuencias biológicas, estableciendo correspondencias entre secuencias de bases para DNA o de aminoácidos para las proteínas. Alinear secuencias es un proceso de importancia básica para los campos de la genómica o la proteómica, siendo de gran utilidad para en el descubrimiento de información funcional, estructural y evolutiva. Por ejemplo, secuencias muy parecidas, o similares, tienen probablemente funciones similares. Adicionalmente, si dos secuencias de diferentes organismos son similares, puede haberse dado una secuencia común ancestral a ambos, lo que permite la derivación de arboles filogenéticos.

Este trabajo tiene como objetivo el análisis 1) de la escalabilidad y 2) del tiempo empleado en cada una de las principales etapas del algoritmo BLAST en función de dos parámetros, la calidad que puede alcanzar una secuencia respecto a una determinada base de datos y el tamaño de la secuencia a buscar. La motivación en el caso de la calidad es que no existen experimentos al respecto en la literatura consultada. Respecto al tamaño de la secuencia, aunque existen datos de escalabilidad[1], creemos necesario la evaluación con tamaños más parecidos a los de las secuencias contenidas en la base de datos.

El articulo se organizará del siguiente modo: en la sección II se introducirá la problemática del alineamiento de secuencias. En la sección III se describirá el algoritmo de BLAST y la implementación del mismo utilizada en la experimentación. En la sección IV se presentarán los resultados. Finalmente se expondrán las conclusiones, sección V, y las nuevas lineas de investigación sugeridas, sección VI.

[1]Dpto. de Arquitectura de Computadores y Sistemas Operativos, Univ. Autónoma de Barcelona, e-mail: {pedro.erencia, alejandro.chacon, juancarlos.moure, antonio.espinosa, porfidio.hernandez} @uab.es

II. Definición de Alineamiento de secuencias.

El alineamiento de secuencias consiste en la comparación de dos (*pair-wise alignment*) o más (*múltiple sequence alignment*) secuencias, buscando conjuntos de caracteres individuales o patrones de caracteres que se encuentran en el mismo orden en las secuencias. Dos secuencias se alinean escribiendose en dos filas. Los caracteres idénticos en la misma columna indican *matches*. Caracteres distintos en la misma columna son *mismatches*. Del mismo modo, un carácter puede estar asociado con un *gap* "-", lo que indica un *indel* (inserción - eliminación). Eliminación e inserción se definen en función de la secuencia que escojamos como aquella contra la que se quiere comparar. Normalmente la fila superior es la secuencia contra la que se alinea y la inferior la que se pretende alinear respecto a la primera. De este modo, si el *gap* se introduce en la fila superior diremos que se ha producido una inserción y si se introduce en la inferior, una eliminación. En la figura 1 se muestra un ejemplo de posible alineamiento. En ella se aprecian *matches* (columnas 2,4,6 y 8), *mismatches* (columna 10), inserciones (columnas 3 y 5) y eliminaciones (columnas 1, 7 y 9).

Formalmente puede definirse el alineamiento del siguiente modo:

Sea Σ un alfabeto, y sean $X = x_1x_2\cdots x_m$ e $Y = y_1y_2\cdots y_n$ dos secuencias sobre Σ. Se extiende Σ añadiendo un símbolo especial "-", que denota un *gap*. Un alineamiento de las secuencias X e Y es una matriz de dos filas cuyas entradas se encuentran en $\Sigma \cup \{-\}$ de modo que,

1. La primera (segunda) fila contiene las letras de $X(Y)$ en orden.
2. Uno o más espacios "-" pueden aparecer entre dos letras consecutivas de Σ en cada fila.
3. Cada columna contiene al menos una letra de Σ.

a	t	-	c	-	t	g	a	t	g
	⋮		⋮		⋮		⋮		
-	t	g	c	a	t	-	a	-	c
1	2	3	4	5	6	7	8	9	10

Fig. 1. Ejemplo de alineamiento. En la parte inferior se indican los números de columna.

La evaluación de la calidad de un alineamiento se realiza mediante el uso de una *matriz de substitución*, que establece una puntuación para cada par del alfabeto. Así, el score para dos secuencias de longitud m y n, donde se han introducido los *gaps* necesarios para que $m = n$ vendría dado por $\sum_{k=0}^{m} sbt(Q[k], D[k])$,

siendo *sbt* la matriz de substitución y Q y D las dos secuencias a alinear. Por ejemplo, para la secuencia de la figura 1, con una matriz de substitución como la de la tabla I la puntuación seria de $-3+5-2+5-3+5-2+5-1-3=6$. Existen diferentes matrices estándar de substitución (PAM o BLOSUM, p.ej) basadas en las probabilidades de substitución de los elementos que las conforman. De un conjunto de posibles alineamientos, se consideran de mayor calidad aquellos con mayor puntuación.

TABLA I
EJEMPLO DE MATRIZ DE SUBSTITUCIÓN

	A	C	G	T	-
A	5	-1	-2	-1	-3
C	-1	5	-3	-2	-4
G	-2	-3	5	-2	-2
T	-1	-2	-2	5	-1
-	-3	-4	-2	-1	*

Para examinar cuan costoso es realizar este proceso de alineamiento podemos representar el mismo tal y como muestra la figura 2[2].

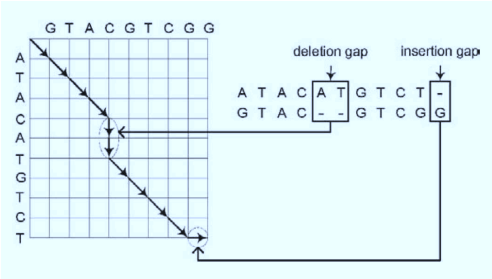

Fig. 2. Alineamiento entre las secuencias ATACATGTCT y GTACGTCGG

En ella, todos los posibles caminos desde el inicio (parte superior izquierda) hasta el final (parte inferior derecha) indican todos las formas posibles de alinear ambas secuencias. Los movimientos hacia abajo y hacia la derecha indican la inserción de gaps en una u otra de las secuencias. Tal cantidad de caminos viene dada por la formula 1

$$a(m,n) = \sum_{k=max(n,m)}^{m+n} \binom{k}{m}\binom{m}{+m-k} \qquad (1)$$

donde m y n son los tamaños de las secuencias. Para $n=m=100$ el número de posibles alineamientos es de $2^{200} \approx 10^{60}$. Si tenemos en cuenta que muchas de las secuencias utilizadas exceden en mucho las 100 bases, por ejemplo el genoma humano, se hace evidente que examinar todos los posibles alineamientos no es una alternativa viable.

III. BLAST

Como respuesta a la necesidad de encontrar un método eficiente para el alineamiento de secuencias se propuso BLAST (Basic Local Alignment Sequence Tool)[3]. El algoritmo utiliza heurísticas para reducir el espacio de búsqueda, centrándose en aquellas regiones en las que es más probable encontrar un buen alineamiento. Esencialmente, BLAST aprovecha el hecho que buenos alineamientos -aquellos con una alta puntuación- , se forman generalmente mediante la unión de múltiples asociaciones.

A. Descripción del algoritmo

BLAST divide la búsqueda en tres etapas.

- *Etapa 1, Identificación de hits.* Definiremos Q como la secuencia a buscar, D como la base de datos de secuencias, cada secuencia contenida en D como S_i, w como el tamaño de la palabra, T como el umbral de *hit* y *sbt* como la matriz de puntuación. Definiremos un *hit* como un par de desplazamientos *(i,j)* para los cuales $\sum_{k=0}^{w-1} sbt(Q[i+k], S_i[j+k] \geq T$.
  El proceso de identificación se realiza del siguiente modo: dada w, se extraen de Q todas las subsecuencias solapadas, de longitud w. Después, se realiza el mismo proceso para cada S_i y se compara, cada una de las subsecuencias de S_i con cada una de las subsecuencias de Q, asignando a la comparación una puntuación, siguiendo el procedimiento descrito en la sección II. Los *hits* serán aquellos pares de subsecuencias cuya puntuación es mayor que T, (vease figura 3). Este procedimiento se repite para todas las S_i.
  Por ejemplo, si tenemos $w = 3$, una secuencia $Q = FIHDK$ y una única secuencia $S = ACDEFGHI$, el conjunto de palabras de Q es $\{FIH, IHD, HDK\}$ y el de S $\{ABC, BCD, CDE, DEF, EFG, GHI.\}$.
- *Etapa 2, Identificación de HSPs (High Scoring Pairs)* Los HSPs son extensiones, sin tener en cuenta *gaps*, en la diagonal d que contenga una pareja de *hits*, o *two-hit*, $(i_1, j_1), (i_2, j_2)$ dentro de una ventana de tamaño A. Por ejemplo, $d = i_1 - j_1 = i_2 - j_2$ y $w \leq i_2 - i_1 \leq A$. Ésta extensión *ungapped* se realiza ampliando el hit en ambas direcciones hasta que el *score* obtenido ha bajado más de X unidades respecto al mejor *score* obtenido hasta el momento (véase figura 4).
- *Etapa 3, Identificación de HSAs (High Scoring Alignments).* A partir de los HSPs reportados por la etapa 2 se realiza una *gapped extension* mediante algún algoritmo de Programación Dinámica, p.ej Smith & Waterman. Estos algoritmos buscan el camino óptimo en la matriz de todos los posibles caminos (véase figura 2).

B. Implementación escogida

La versión de BLAST utilizada es la desarrollada por el NCBI en su versión 2.2.26. La aplicación está escrita es C++ (con algunas secciones en C para el núcleo de los algoritmos) y su código puede descargarse libremente. Su diseño es altamente modular y orientado a un fácil mantenimiento, separando las

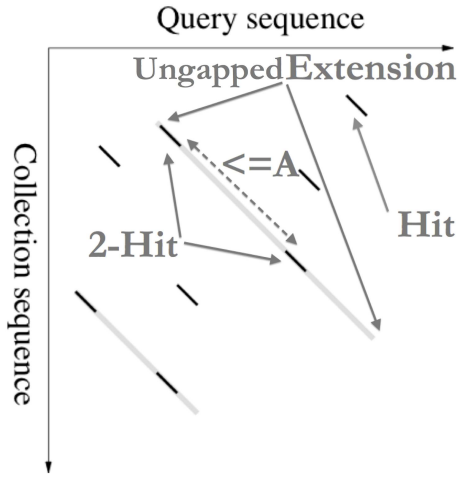

Fig. 3. Etapas 1 y 2 de BLAST. En la primera etapa se identifican los hits, en color más oscuro. En la etapa 2 aquellas parejas de hits que están en la misma diagonal y a una distancia menor o igual a un determinado umbral son unidos y extendidos.

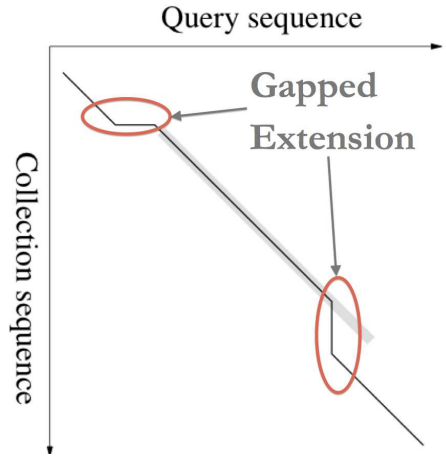

Fig. 4. Etapa 3 de BLAST. Sobre el HSP obtenido en la etapa 2 se realiza el alineamiento final con gaps, representados por las lineas horizontales y verticales.

diferentes aplicaciones en proyectos independientes, lo que ha facilitado la compilación únicamente de aquella aplicación en la que se ha trabajado (*blastp*).

B.1 Estructuras de datos

Las etapas 1 y 2 acceden con frecuencia a tres estructuras de datos. La primera es la *Lookup Table* que mantiene la correspondencia entre cada palabra de tamaño w y sus posiciones en Q.
Para implementar esta etapa BLAST preprocesa Q del siguiente modo: para cada posición i de Q se calculan el conjunto de sus vecinos $N(Q[i...i+w-1], T)$ como todas aquellas palabras de longitud w para las que $\sum_{k=0}^{w-1} sbt(Q[i+k], D[j+k] \geq T$. Esta información se guarda en una tabla de acceso directo con $|\Sigma|^w$ entradas, una por cada posible palabra de longitud w. Cada una de las entradas contiene las posiciones en Q donde se halla esa palabra o alguna de sus vecinas. En la figura 5 se muestra un ejemplo de tabla construida según el procedimiento descrito. La *Lookup table* mantiene dicha tabla en un vector, al que se refiere como *backbone*. Cada una de las entradas son 4 enteros, indicando el primero la cantidad de *hits* de esa palabra y de sus vecinas en Q. Los otros tres indican las posiciones en Q de los tres primeros *hits* para esa palabra. En el caso que la palabra tenga más de 3 hits, la primera posición contendrá un apuntador a un vector con las posiciones en Q [4].

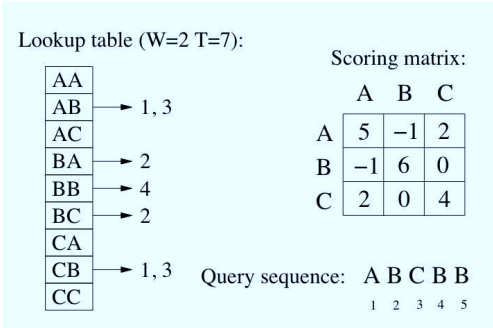

Fig. 5. Lookup table. En este ejemplo, el alfabeto contiene 3 caracteres, A, B y C. La tabla de la izquierda se construye utilizando la *matriz de puntuación* y la secuencia de entrada Q de la derecha con un tamaño de palabra $w = 2$ y un umbral $T = 7$. Cada entrada tiene asociada una lista de posiciones en Q.

Además del *backbone*, la tabla tiene asociado un vector de bits, llamado *vector de presencia*, con tantas posiciones como posibles palabras de longitud w, o entradas en la tabla. Cada bit está o no activado en función de si esa palabra o alguna de sus vecinas tiene alguna ocurrencia en Q. Para las proteínas el alfabeto es de 24, con 5 bits para codificar cada carácter. El tamaño del *backbone*, en cuanto a entradas, es por lo tanto de 32^3. Suponiendo enteros de 4 Bytes, cada entrada son 4 Bytes $\times$ 4 contadores = 16 Bytes, lo que implica un tamaño de 512 KB, al que hemos de sumar el *vector de presencia*, unos 4K.

La tercera estructura es el vector de diagonales, encargado de guardar, para cada diagonal, el último *hit* encontrado. Esta información es necesaria para realizar la extensión *ungapped* de la etapa 2 en caso de que se encuentre un nuevo *hit* en la misma diagonal a una distancia menor que el umbral determinado. Este vector tiene tantas entradas como bases Q. Suponiendo igualmente un entero de 4 Bytes por entrada, esto supone $4 \times |Q|$ Bytes.

Resumiendo, podemos estimar el tamaño conjunta de estas estructuras en función de Q como $(516 \times 1024) + (4 \times |Q|)$ Bytes.

B.2 Distribución del trabajo entre Threads

Cuando se busca una única secuencia, la distribución entre los Threads se realiza tal y como muestra el algoritmo 1. En el mismo, la etapa 1 se realiza en la linea 4, la etapa 2 en la linea 7 y la etapa 3 en la linea 11.

Las secuencias que forman la base de datos son asignadas a cada Thread mediante el acceso de éstos a una estructura compartida protegida con una sección crítica. Por lo tanto, cada Thread alinea la secuencia a buscar con una secuencia de la base de

Algoritmo 1 Trabajo realizado por cada Thread

```
1:  mientras haya secuencias en bbdd hacer
2:      obtener secuencia
3:      para todo chunk hacer
4:          hits ← ScanLookupTable
5:          para todo hit hacer
6:              si hit previo diagonal y distancia ≤ A
    entonces
7:                  HSPs ← UngappedExtension
8:              fin si
9:          fin para
10:         para todo HSP hacer
11:             HSAs ← GappedExtension
12:         fin para
13:     fin para
14: fin mientras
```

datos de forma completa, esto es, realizando las tres etapas de forma secuencial.

Los llamados *chunks* que aparecen en el algoritmo son secciones de la secuencia. Antes de realizar la búsqueda, BLAST aplica un enmascaramiento, tanto a la secuencia a buscar como a la base de datos. Este proceso consiste en la eliminación o desestimación de regiones de *baja complejidad* o con repeticiones intercaladas, que hacen *match* en un gran número de secuencias.

IV. Resultados

Como base de datos donde buscar se utilizo SWISS-PROT. A la hora de escoger las secuencias a buscar se identificaron los dos parámetros mencionados en la sección I. Respecto al tamaño de la secuencia, se estudiaron las longitudes de las secuencias contenidas en SWISS-PROT. De éstas, únicamente un 3,52 % tienen una longitud mayor de 1000 bases. Del resto, el 66.31 % tienen un valor menor de 400.

En base a estos datos se escogieron 200 y 1000 como las longitudes con las que experimentar. La de 200 por ser un tamaño representativo y la de 1000 por ser un caso menos frecuente pero con mayor carga de trabajo.

El segundo parámetro a considerar era la calidad de los alineamientos que una determinada secuencia podía tener respecto a SWISS-PROT, o *E-Value*. Este valor describe el número de *hits* que puede esperarse encontrar por azar cuando se busca una base de datos de un determinado tamaño. Su valor disminuye exponencialmente a medida que el *Score* aumenta. A menor *E-Value*, o mientras más cercano a 0, más significativo es un *match*. Para determinar este parámetro se utilizó el valor de *E-Value* que BLAST proporciona para cada búsqueda.

En la tabla II se muestran las características de las secuencias seleccionadas, junto con los nombres con los que se las designará durante el resto del trabajo.

Las pruebas se realizaron sobre un equipo con procesador *i7 920* a 2.66 GHZ (máximo sin TurboBoost) con 6 GB de RAM. Este procesador consta de 4 cores, con dos Threads por core. Cada core tiene una caché de nivel 1 de 32KB y una de nivel 2 de 256 KB., ambas dedicadas. Existe un tercer nivel de caché, de 8 MB, compartido por todos los núcleos.

TABLA II
Secuencias a buscar

Nombre	Tamaño	*E-Value*
200A	227	3e-111
200B	210	9.7
1000A	1033	7e-160
1000B	1033	6.8

A. Impacto en la escalabilidad en función de la calidad

Para realizar la comparación se utilizaron las dos secuencias de 1000, con los *E-Value*s mostrados en la tabla II. Como puede observarse en la figura 6 la diferencia entre buscar una secuencia con un *E-Value* alto y otra con un *E-Value* bajo es muy pequeña. La mayor diferencia se da para 1 Thread, alcanzando para las secuencias con mayor *E-Value* un incremento cercano el 7 %. Esto puede deberse al hecho de que, al seleccionar las secuencias a buscar, hemos utilizado secuencias que, aunque con una diferencia significativa en cuanto al *E-Value*, han pasado por todas las etapas. Estas secuencias han generado un HSP con un *score* lo suficientemente alto como para ser extendidas en la tercera fase, e incluso en ésta han superado el umbral, convirtiendose en HSAs, aunque con un *score* muy bajo. Por lo tanto, es de suponer que al menos buena parte del trabajo ha sido realizada aunque el *score* sea bajo.

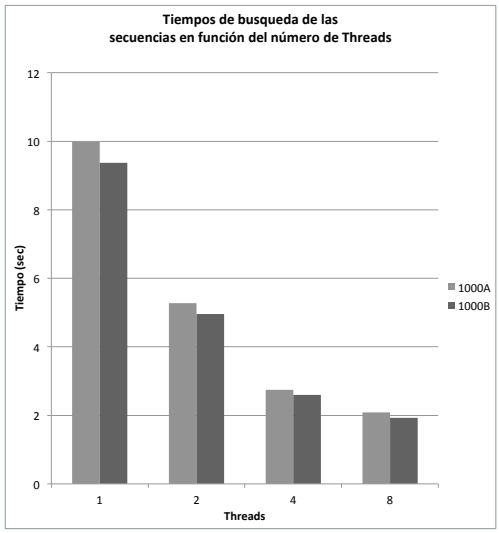

Fig. 6. Tiempos de ejecución para secuencias con diferente calidad

B. Impacto en el tiempo de las diferentes etapas en función de la calidad

Habiendo identificado las funciones correspondientes a cada etapa (trabajo necesario para la construcción del pseudo-código del algoritmo 1), se utili-

zo la herramienta de *profiling* OProfile para caracterizar el tiempo empleado en cada una de ellas. OProfile es *profiler* para sistemas Linux con un bajo *overhead*, capaz de realizar el *profiling* de cualquier código en ejecución utilizando directamente los contadores H/W. El evento utilizado para medir fue *CPU_CLK_UNHALTED*. En la tabla III se indican el nombre de las funciones en la implementación correspondientes a cada una de las fases.

TABLA III
FUNCIONES ASOCIADAS A LAS ETAPAS

Nombre Función	Etapa
BlastSmallAaScanSubject	1
BlastAaWordFinder + BlastAaExtendTwoHit	2
SemiGappedAlign + RestrictedGappedAlign	3

Los resultados se muestran en la figura 7. En todos las pruebas el número de ciclos gastados por las 5 funciones de la tabla III suponían aproximadamente el 95 % de los ciclos consumidos por toda la aplicación, lo que resulta coherente, puesto que son las 3 etapas fundamentales del algoritmo.

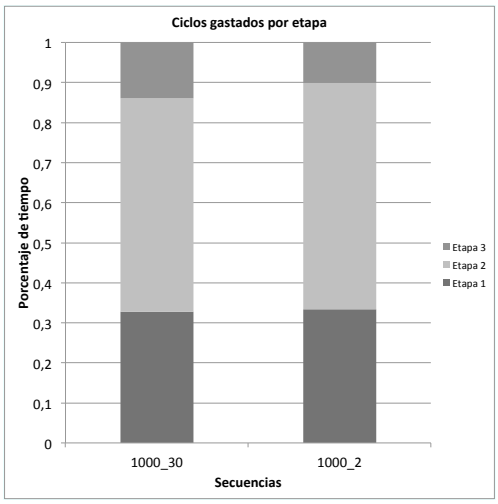

Fig. 7. Porcentajes de tiempo de cada etapa para secuencias con diferente calidad

Se puede apreciar como el tiempo ahorrado en el alineamiento de la secuencia con el bajo *E-Value* viene en mayor proporción por el ahorro en la tercera etapa. Esto corrobora nuestra sospecha según la cual buena parte del trabajo ha sido realizado igualmente aunque el *score* sea bajo. Teniendo en cuenta el menor impacto de la tercera fase, una ligera reducción en la misma explica el escaso impacto de la calidad en el tiempo total del alineamiento.

C. Impacto en la escalabilidad en función del tamaño

En la figura 8 se muestra el speedup para las diferentes secuencias y combinaciones de Threads.

Para 2 y 4 Threads los speedup son cercanos a los ideales, lo que indica una buena escalabilidad. Incluso para 8 Threads, con 2 Threads por core, hay mejora, llegando en el caso de la secuencia 1000B a un speedup cercano a 5.

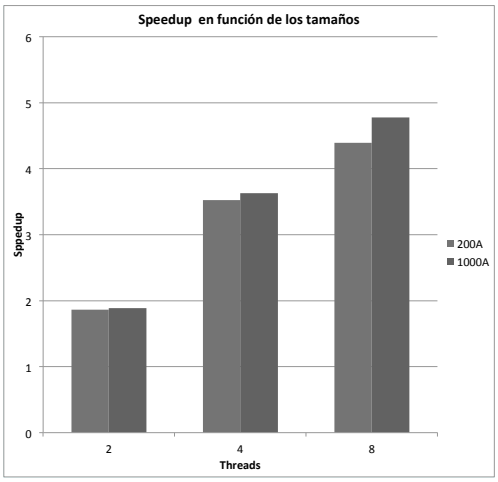

Fig. 8. Speedup para secuencias con diferente tamaño

El hecho de que haya mejora con 8 Threads es indicativo de una limitación de latencias, quizás en relación al acceso a la *Lookup Table*. Tal y como se comentó en la sección III-B.1, hay 2 estructuras utilizadas en la primera etapa de BLAST, el *backbone* y el *vector de presencia*. El segundo ocupa 4 KB y se accede para cada *w-palabra* de la base de datos, por lo que probablemente resida continuamente en la caché de nivel 1. En cambio, el *backbone* ocupa 512 KB, tamaño muy superior a los 256 KB de la caché de datos de nivel 2 del *i7*. Si en lugar de utilizar para la creación de la tabla los 32 posibles caracteres generables a partir de los 5 bits necesarios para codificar los 24 caracteres del alfabeto, se utilizasen únicamente esos 24, pasaríamos de $32^3 \times 16 = 512$ KB a $24^3 \times 16 = 216$ KB, lo que si cabría en la L2. Incluso si se quisieran utilizar 28 caracteres para el alfabeto, para secuencias a buscar menores de 64 Kbases podrían utilizarse 2 Bytes en lugar de enteros de 4, con lo que tendríamos $28^3 \times 8 = 171,5$ KB. De hecho, BLAST permite utilizar 2 Bytes, pero mantiene las 32^3 entradas, con lo que volvemos a superar la L2 con $32^3 \times 8 = 262$ KB. Ciertamente, utilizar 32 tiene como ventaja que los cálculos para resolver el indice en la tabla son más rápidos y pueden realizarse con *shifts* de bits. Sin embargo, habría de evaluarse si esa reducción en el computo compensa el acceso continuo a la L2, especialmente teniendo en cuenta que el acceso a la *Lookup Table* es aleatorio, ya que las *w-palabras* de la base de datos no tienen porque seguir ningún patrón para resolverse en posiciones de la tabla cercanas entre si.

D. Impacto en el tiempo de las diferentes etapas en función del tamaño

Dos observaciones pueden extraerse de la figura 9. En primer lugar el tiempo dedicado a las fases 1 y 2 es mucho mayor que para la fase 3. Para la secuencia

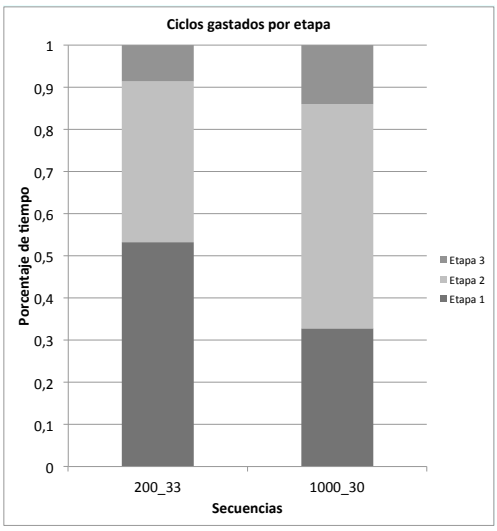

Fig. 9. Porcentajes de tiempo de cada etapa para secuencias con diferente tamaño

de 200 la etapa 1 ocupa aproximadamente el 50 %, la etapa 2 el 35 % y la etapa 3 sobre el 10 % (respecto al tiempo total de la aplicación). Por lo tanto las etapas 1 y 2 representan más del 80 % del tiempo de ejecución. Para la secuencia de 1000 las etapas 1 y 2 también representan la mayor parte del tiempo. Sin embargo, las etapas 1 y 2 han intercambiado sus porcentajes, 32 % y 50 %. Los aproximadamente tres puntos que pierde la etapa 1 son los que incrementa la etapa 3, con un 13 %. Esta diferencia entre las secuencias de 1000 y 200 puede explicarse por la mayor cantidad de *hits* que se encuentran en la secuencia de 1000. Este incremento implica un mayor trabajo en la identificación de *hits* dentro de la misma diagonal y, en caso de hallarse, un aumento de las extensiones, tanto *ungapped* como *gapped*.

V. Conclusiones

En este trabajo se ha examinado el impacto de la calidad y del tamaño de la secuencia a buscar sobre la escalabilidad de BLAST y el tiempo empleado en cada una de las etapas del algoritmo. En relación a la calidad, se ha podido comprobar como menor calidad implica menores tiempos de ejecución, aunque muy ligeramente. Del mismo modo, esta ligera reducción se produce en mayor proporción en la tercera etapa del algoritmo.

En cuanto al impacto del tamaño, se ha comprobado como la escalabilidad de la implementación analizada se acerca a los valores ideales para 2 y 4 núcleos, independientemente del tamaño de la secuencia. Con 8 núcleos y SMT (Simultaneous Multi-Threading) se consigue también mejora, aunque de una magnitud inferior. Se ha comprobado también como el incremento en el tamaño implica incrementos significativos en las etapas 2 y 3 del algoritmo.

VI. Trabajo Futuro

En cuanto al análisis de escalabilidad se propone realizar nuevos experimentos en máquinas con mayor número de núcleos para estudiar el límite la mejora obtenible mediante el SMT para diferentes tamaños.

Igualmente, resultaría interesante estudiar las implementaciones de BLAST para otros entornos, concretamente para GPUs. El auge de este tipo de hardware masivamente paralelo para la aceleración de algoritmos científicos ha llevado a la creación de diferentes implementaciones de BLAST. En concreto, la versión auspiciada por Nvidia [5] informa de un *speedup* de 6 para una única GPU Tesla. El hecho de que, tal y como se ha mostrado, el uso del SMT consigue mejoras cercanas a 5 puede significar, o bien una limitación intrínseca en la forma actual del algoritmo, o una implementación susceptible de mejora en GPUs (o una combinación de ambas explicaciones). En este sentido el reto sería la consecución de un esquema de paralelización eficiente del algoritmo para ese tipo de H/W.

En relación con lo anterior, el uso de un H/W distinto puede además llevarnos a la consideración de otros factores, más allá del tiempo de ejecución. Uno de ellos, que cada día adquiere mayor visibilidad, es el consumo energético. Evaluar que alternativa es no solo más rápida, sino más eficiente energéticamente, puede resultar interesante para entornos en los que este parámetro sea determinante.

Agradecimientos

El presente trabajo ha sido financiado mediante el proyecto MICINN-Spain TIN2011- 28689-C02-01.

Referencias

[1] Christiam Camacho, George Coulouris, Vahram Avagyan, Ning Ma, Jason Papadopoulos, Kevin Bealer, and Thomas L Madden, "Blast+: architecture and applications," *BMC Bioinformatics*, vol. 10, no. 1, pp. 421, 2009.

[2] K.M. Chao and L. Zhang, *Sequence Comparison: Theory and Methods*, Computational Biology. Springer, 2008.

[3] S F Altschul, T L Madden, A A Schaffer, and J Zhang, "Gapped blast and psi-blast: a new generation of protein database," *Nucleic Acids Research*, vol. 25, no. 17 (Sep), pp. 3389–3402, 1997.

[4] Michael Cameron, Hugh E Williams, and Adam Cannane, "A deterministic finite automaton for faster protein hit detection in blast.," *Journal of computational biology a journal of computational molecular cell biology*, vol. 13, no. 4, pp. 965–978, 2006.

[5] Weiguo Liu, Bertil Schmidt, and Wolfgang Muller-Wittig, "Cuda-blastp: Accelerating blastp on cuda-enabled graphics hardware.," *IEEEACM transactions on computational biology and bioinformatics IEEE ACM*, vol. 28, no. 5, pp. 881–5, 2010.