Universitat Autònoma de Barcelona

escola d'enginyeria

# 3904 - Optimització d'una aplicació bioinformàtica d'alineament de seqüències executada en processadors multi-core i many-core (GPUs)

**Memòria del Projecte Fi de Carrera**
d'Enginyeria en Informàtica
realitzat per
*Carles Figuera Penedo*
i dirigit per
*Juan Carlos Moure López*
Bellaterra, 21 de Juny de 2011

Universitat Autònoma de Barcelona

escola d'enginyeria

El sotasignat, *Juan Carlos Moure López*,

Professor/a de l'Escola Tècnica Superior d'Enginyeria de la UAB,

**CERTIFICA**:

Que el treball a què correspon aquesta memòria ha estat realitzat sota la seva direcció per en *Carles Figuera Penedo*.

I per tal que consti firma la present.

Signat:

Bellaterra, 21 de Juny de 2011

# Índice de contenido

# Capítulo 1

# Introducción

Las aplicaciones bioinformáticas de búsqueda de homología entre secuencias están relacionadas con el estudio de los seres vivos. Todo organismo vivo contiene un código genético formado por largas secuencias de caracteres de ADN. Una de las tareas más importantes de las aplicaciones de alineamiento de secuencias es estudiar la homología de estas secuencias, que se refiere a la situación en la que las secuencias (tanto de ADN como de proteínas) son similares entre sí debido a que presentan un mismo origen evolutivo.

Para encontrar las similitudes de una secuencia hay que compararla con una base de datos de secuencias conocidas llamadas bancos de secuencias. Estos bancos son actualizados cada pocas semanas y una de las principales problemáticas que presentan es su importante crecimiento durante las últimas décadas (figura 1.1), trayendo consigo la necesidad de disponer de sistemas computacionales que logren procesar este gran volumen de datos y generar resultados de la forma más rápida posible.

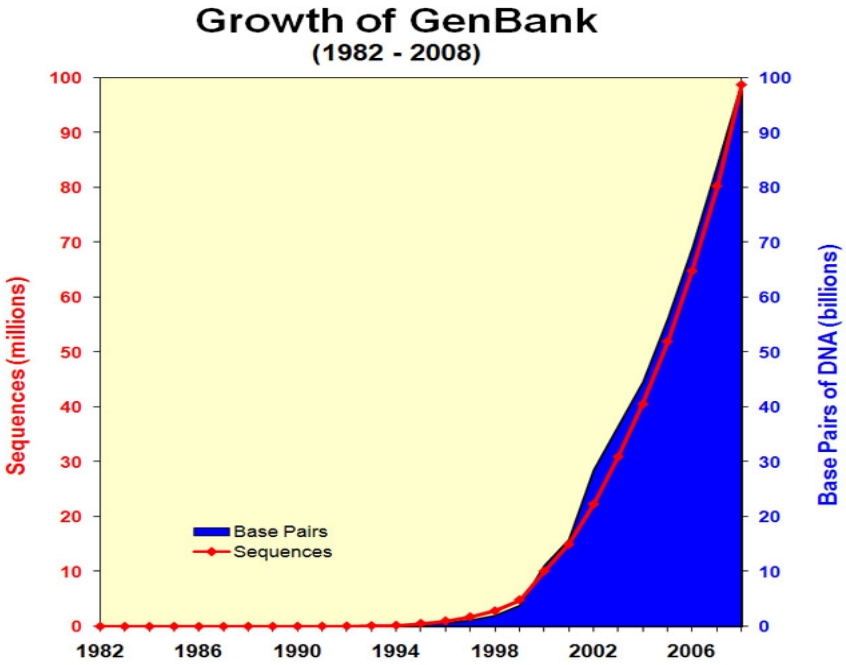


***Fig. 1.1:*** *Crecimiento de los datos del banco GenBank*

BLAST (Basic Local Alignment Search Tool) es un conjunto de aplicaciones informáticos de alineamiento y homología de secuencias, ya sea de ADN o de proteínas. Solo la versión web de BLAST tiene más de 100.000 consultas sobre ADN cada día. El programa es capaz de comparar

una secuencia problema (también denominada *query*) contra una gran cantidad de secuencias que se encuentren en una base de datos. El programa encuentra las secuencias de la base de datos que tienen mayor parecido a la secuencia problema. Uno de los problemas que presenta el uso de este tipo de aplicaciones es la gran cantidad recursos computacionales que consumen (volumen de datos a procesar, volumen de datos a leer de disco, etc.) que provoca que el tiempo de respuesta de estas aplicaciones sea muy grande.

Para hacer frente a esta gran cantidad de cómputo, estas aplicaciones hacen uso de sistemas de cómputo paralelo como los procesadores multi-core, que combinan dos o más núcleos, y que gracias a ellos se pueden ejecutar estas aplicaciones en múltiples hilos de ejecución independientes, de forma que es posible la concurrencia. Es decir, se pueden ejecutar al mismo tiempo en paralelo.

Pero además, hay que estudiar la posibilidad de usar tecnologías como CUDA. Se trata de una plataforma software que intenta explotar las ventajas de las GPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.

## 1.1 Objetivos del proyecto

Los objetivos de este proyecto son:

- Analizar el algoritmo de una aplicación BLAST existente: *blastp,* versión BLAST para proteínas.
- Analizar el comportamiento del programa según los parámetros de entrada usados.
- Detectar las partes del código que consumen más tiempo de ejecución, tanto en su versión para un procesador como en la versión multi-core.
- Identificar los problemas de rendimiento en las partes del código que consumen más tiempo de ejecución.
- Estudiar los problemas de rendimiento encontrados y optimizarlos, tanto en la versión para un procesador como en la versión multi-core.
- Encontrar líneas futuras de investigación para aportar más conocimiento al análisis de rendimiento o bien para la optimización de la aplicación.

## 1.2 Planificación temporal del trabajo

A continuación se muestran tanto la planificación temporal inicial como la final de este proyecto.

La planificación del proyecto inicial del proyecto presentaba 5 fases que se mencionan a continuación:

- Conocimientos previos: adquisición de conocimientos biológicos necesarios para la correcta realización del proyecto.

- Análisis del rendimiento: estudio del rendimiento de Blastp (tiempo, ciclos, instrucciones, ...) y análisis del efecto de los parámetros de entrada en dicho rendimiento.

- Optimización para versión multi-core: diseño e implementación de las optimizaciones estudiadas para el uso de procesadores multi-core. Además, análisis del rendimiento de la optimizaciones.

- Implementación many-core (CUDA): estudio del modelo y arquitectura CUDA. Diseño e implementación del código para el uso con CUDA. Análisis del nuevo rendimiento.

- Documentación: realización de informe previo y de una memoria final de proyecto.

A continuación, se muestra un diagrama de Gantt de la planificación temporal inicial que se estableció para el proyecto:

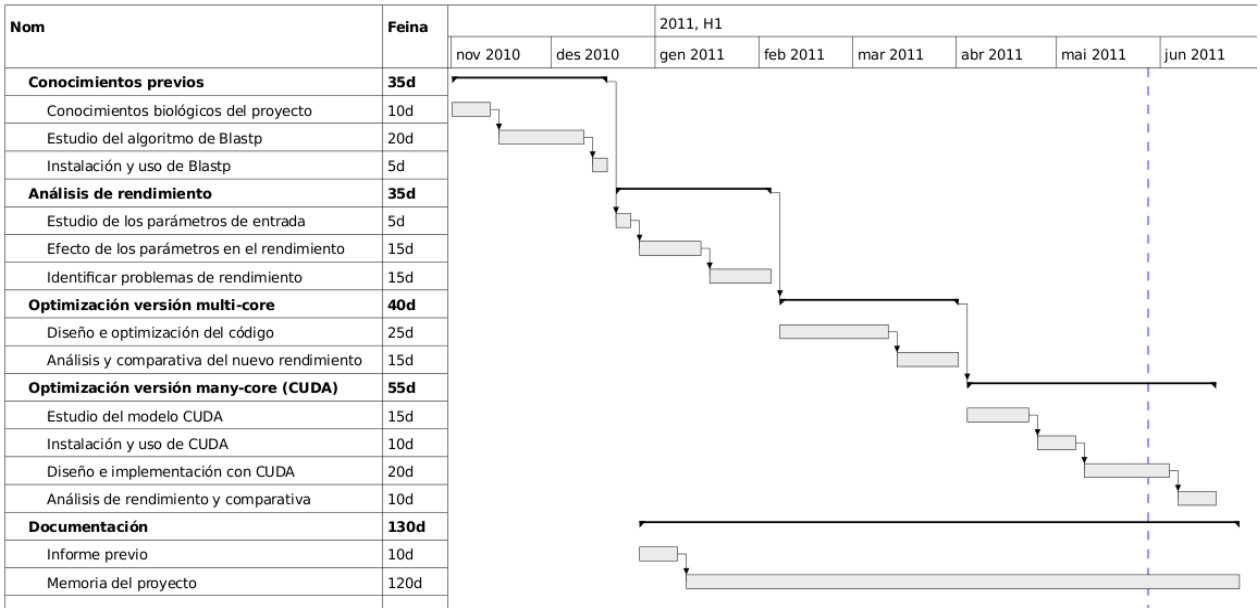


***Fig. 1.2:*** *Daigrama de Gantt de la planificación temporal inicial*

Además de un análisis de rendimiento y de la implementación de optimizaciones en un sistema multi-core, la planificación inicial también presentaba la implementación y el análisis de la aplicación usando una arquitectura CUDA. A causa del tiempo dedicado de más a la complejidad del análisis y a la optimización de *blastp*, se ha tenido que suprimir esta fase quedando la planificación de la siguiente manera:

- Conocimientos previos: adquisición de conocimientos biológicos necesarios para la correcta realización del proyecto.

- Análisis del rendimiento: estudio del rendimiento de *Blastp* (tiempos de respuesta, número de ciclos, número de instrucciones, fallos de caché,...) y análisis del efecto de los parámetros de entrada en dicho rendimiento.

- Optimización para un solo procesador: diseño e implementación de las optimizaciones estudiadas para el uso de sistemas con un solo procesador. Además, realización de un análisis del rendimiento de dichas optimizaciones.

- Optimización para versión multi-core: diseño e implementación de las optimizaciones estudiadas para el uso de procesadores multi-core. También realización de un análisis del rendimiento de dichas optimizaciones.

- Documentación: realización de informe previo y de una memoria final de proyecto.

A continuación, se muestra un diagrama de Gantt de la planificación temporal final para el proyecto:

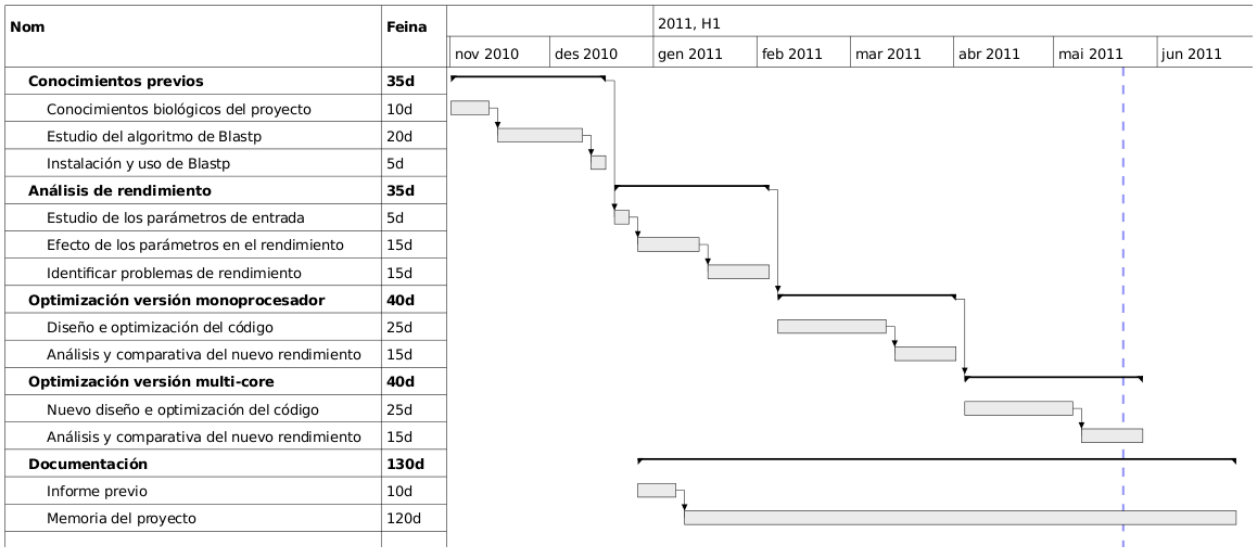


***Fig. 1.3:*** *Daigrama de Gantt de la planificación temporal final*

## 1.3 Organización de la presente memoria

El siguiente esquema muestra la estructura del presente documento, dividiéndose en los siguientes capítulos:

- Capítulo 2. Conceptos previos: introducción a conceptos necesarios para la realización del proyecto. Se centra tanto en la descripción de conceptos previos de la biología (proteínas, aminoácidos,...) como en el funcionamiento del conjunto de aplicaciones BLAST.

- Capítulo 3. Análisis de rendimiento: análisis de rendimiento de la aplicación *blastp*. Para encontrar las causas de un rendimiento bajo en una aplicación es necesario estudiar y entender cómo se estructura y cómo funciona.

- Capítulo 4. Optimizaciones: incluye las distintas optimizaciones que han sido estudiadas e implementadas y sus resultados obtenidos. Estas optimizaciones se pueden clasificar en dos tipos: para sistemas con un único procesador y para sistemas multi-core.

- Capítulo 5. Conclusiones y líneas futuras: incluye las conclusiones y las líneas futuras de investigación, donde se nombran algunas otras propuestas de optimización.

- Capítulo 6. Bibliografía: enumera la bibliografía usada para llevar a cabo el proyecto.

# Capítulo 2

# Conceptos previos

El objetivo de este capítulo es introducir los conceptos previos necesarios para la realización y comprensión del proyecto. Se centra tanto en la descripción del marco teórico, de conceptos previos de la biología (proteínas, aminoácidos,...) como en el funcionamiento del conjunto de aplicaciones BLAST. También se describen las diferentes herramientas usadas en la búsqueda de homología de secuencias, centrándonos en *Blastp*, caso de estudio en este proyecto.

## 2.1 Marco Teórico

Se describen el procesador clásico y conceptos asociados, para explicar posteriormente los procesadores multi-core y multi-thread, así como los problemas para conseguir alto rendimiento en estos sistemas.

### 2.1.1 Procesador clásico

El proceso de ejecución de una instrucción se puede dividir en cinco etapas:

1. Búsqueda de la instrucción (*fetch*)
2. Decodificación de la instrucción (*decode*)
3. Ejecución de la instrucción (*execute*)
4. Acceso a memoria (*memory*)
5. Escritura del resultado (*writeback*)

En la siguiente imagen (figura 2.1) se muestra la ejecución secuencial de dos instrucciones en un procesador clásico. Se puede apreciar que hasta que la primera instrucción no finaliza, no se puede comenzar la búsqueda de la segunda instrucción.


***Fig. 2.1:*** *Ejemplo ejecución en un procesador clásico*

De este modo, el tiempo de ejecución de un programa se puede expresar con la siguiente ecuación:

$$T = N \cdot CPI \cdot t$$

donde :

- N = número de instrucciones de un programa.
- CPI = número medio de ciclos por instrucción.
- t = tiempo de un ciclo de instrucción.

Para reducir el tiempo de ejecución en esta ecuación independientemente del nivel de integración, existen dos alternativas: reducir el número de instrucciones y/o reducir el CPI. Existen dos tipos de filosofías en las arquitecturas de procesadores que atacan este problema:

- RISC (Computadoras con un conjunto de instrucciones reducido): Implementación H/W más simple, por tanto más rápida y eficiente. Se basa en disponer de un repertorio de instrucciones reducido, permitiendo su implementación por hardware. Los programas tendrán un número elevado de instrucciones pero prácticamente la totalidad de ellas se ejecutarán en un ciclo de reloj.

- CISC (Computadoras con un conjunto de instrucciones complejo): Implementación H/W más compleja y por tanto más lenta e ineficiente. Se basa en disponer de un repertorio de instrucciones amplio y complejo. El número de instrucciones de un programa es menor que en el RISC, pero el CPI suele ser mayor.

### 2.1.2 Procesamiento Segmentado (*pipelined*)

En el procesamiento segmentado se adopta una nueva estrategia con el objetivo de disminuir el tiempo medio de ejecución por instrucción de una aplicación. Se divide internamente el computador en segmentos individuales, cada uno especializado en una de las etapas.

A diferencia del procesador clásico donde todas las etapas tenían que completarse antes de buscar la instrucción siguiente, ahora la existencia de segmentos especializados permite el solapamiento en la ejecución de las instrucciones. Así, un segmento puede empezar a trabajar con una nueva instrucción sin la necesidad de que la instrucción anterior haya finalizado todas las etapas.

El resultado es un aumento del número de instrucciones ejecutadas por ciclo. Como muestra la figura 2.2, con la ejecución segmentada de instrucciones, se puede llegar a ejecutar una instrucción por ciclo.

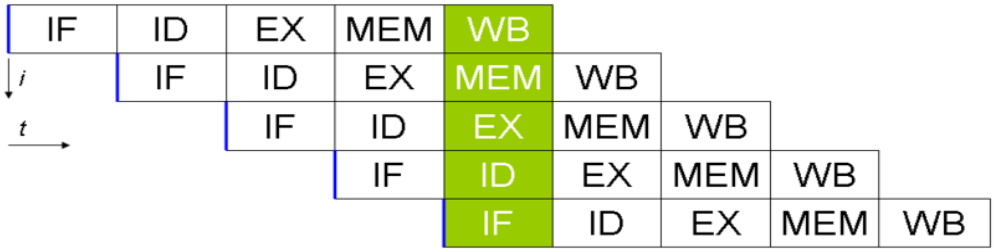


***Fig. 2.2:*** *Ejemplo ejecución en procesador pipelined*

### 2.1.3 Procesador superescalar

Un procesador superescalar es capaz de ejecutar más de una instrucción en cada etapa del pipeline del procesador. El número máximo de instrucciones en cada etapa depende del número y del tipo de las unidades funcionales de que disponga el procesador.

Sin embargo, un procesador superescalar sólo es capaz de ejecutar más de una instrucción simultáneamente si las instrucciones no presentan ningún tipo de dependencia. Las dependencias que pueden aparecer son:

- Dependencias estructurales: cuando dos instrucciones requieren el mismo tipo de unidad funcional pero su número no es suficiente.

- Dependencias de datos: situación en que las instrucciones de un programa se refieren a los resultados de otras anteriores que aún no han sido completadas. Se clasifican en:

  - RAW (*Read After Read*): situación donde se necesita un dato que aún no ha sido calculado.
  - WAR (*Write After Read*): una instrucción necesita escribir en un registro sobre el que otra instrucción previamente debe leer.
  - WAW (*Write After Write*): una instrucción necesita escribir en un registro sobre el que otra instrucción previamente debe escribir.

- Dependencias de control: cuando existe una instrucción de salto que puede variar la ejecución de la aplicación.

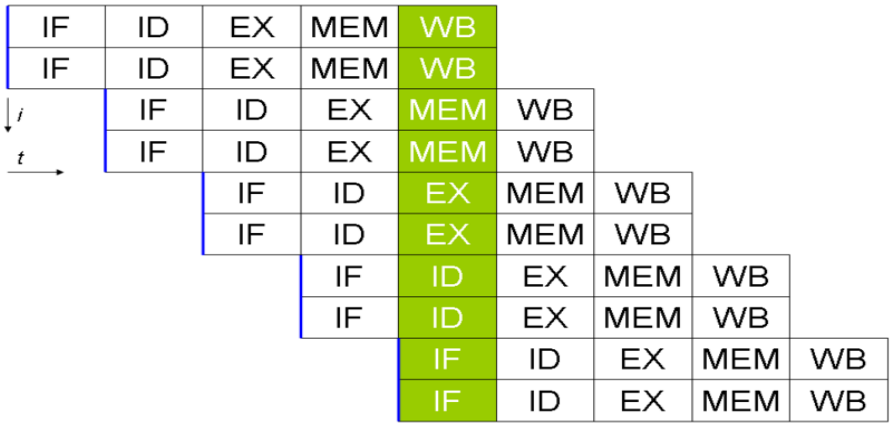


***Fig. 2.3:*** *Ejemplo de ejecución superescalar*

Podemos distinguir diferentes tipos de procesadores por la forma de actuar ante una dependencia estructural o de datos:

- Procesador con ejecución en orden: las instrucciones quedarán paradas a la espera de que se resuelva la dependencia.

- Procesador con ejecución fuera de orden: las instrucciones dependientes quedarán paradas pero será posible solapar parte de la espera con la ejecución de otras instrucciones independientes que vayan detrás.

En el caso de las dependencias de control, se conoce como ejecución especulativa de instrucciones a la ejecución de instrucciones posteriores a la instrucción de salto (antes de que el PC llegue a la instrucción de salto).

### 2.1.4 Limitaciones del procesador single-thread

A continuación describiremos una serie de factores que limitan el rendimiento de la ejecución de una aplicación en un procesador single-thread.

- *Problema de la memoria*

La diferencia de velocidad entre procesador y memoria, limita el rendimiento del procesador. Las operaciones de memoria son lentas comparadas con la velocidad del procesador. Los accesos a memoria, por ejemplo en un fallo de cache, pueden consumir de 100 a 1000 ciclos de reloj, durante los cuales el procesador debe esperar a que el acceso a memoria finalice. Por tanto, un aumento de la frecuencia de reloj del procesador sin incrementar la velocidad de la memoria solamente mejoraría el rendimiento en un pequeño porcentaje. Los ciclos de cómputo se realizarían más rápido pero el tiempo de acceso a memoria continuaría siendo el mismo. Esto se puede apreciar en la siguiente imagen (figura 2.4) que representan las fases de ejecución de un programa single-thread en dos procesadores con distinta frecuencia de reloj.

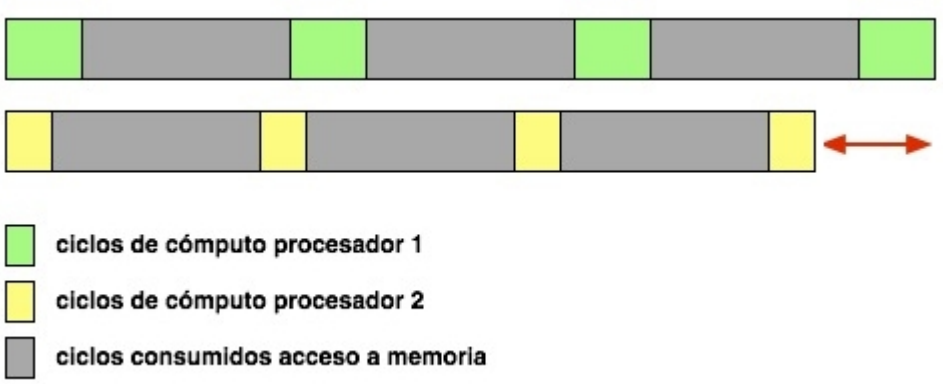


***Fig. 2.4:*** *Ejemplo de fases de ejecución de un programa en dos procesadores de distinta frecuencia*

Además hay que tener en cuenta que a los cientos de ciclos que se consumen en cada acceso a memoria hay que sumarle decenas de ciclos extra, por cada acceso a nuevos niveles de cache (provocados por fallos en el nivel anterior). La solución para aprovechar los ciclos en los que el procesador esta esperando a que finalice la operación de memoria es el multi-threading por hardware. El multi-threading por hardware es una propiedad que permite al procesador alternar de un thread a otro thread cuando el thread que esta ocupando el procesador queda parado. Esta solución se analizará en profundidad más adelante.

- *Calor y coste asociado*

El incremento de la frecuencia de reloj del procesador implica un aumento de la potencia consumida y del calor generado. En la actualidad los altos valores de frecuencia de reloj de los procesadores suponen un problema, tanto económico (consumo eléctrico, y gasto dedicado a la

disipación del calor y refrigeración), como tecnológico (dificultad para disipar la gran cantidad de calor generado en la superficie de un procesador). Por estos motivos, se abandona la idea de aumentar la frecuencia de reloj del procesador para aumentar el rendimiento, y se opta por añadir más procesadores en el mismo chip. Con esta solución el calor se incrementa de forma lineal y no exponencial como ocurre con el aumento de frecuencia de reloj.

### 2.1.5 Multi-threading

Para entender conceptos como el multi-threading por software, por hardware y las ventajas de los procesadores multi-thread es necesario conocer la diferencia entre thread y proceso.

*Threads y Procesos*

Un proceso es secuencia de código ejecutable en ejecución. Cada proceso posee un espacio de direccionamiento propio para almacenar sus estructuras de datos asociados. Un proceso esta formado por uno o más threads o hilos de ejecución.

Un hilo es la unidad mínima de procesamiento. Los hilos existen dentro de un proceso y comparten recursos como el espacio de memoria, la pila de ejecución y el estado de la CPU. Un proceso con múltiples hilos tiene tantos flujos de control como hilos. Cada hilo se ejecuta con su propia secuencia de instrucciones de forma concurrente e independiente.

Las ventajas de realizar la concurrencia a nivel de hilo en lugar de a nivel de proceso son varias. Los hilos se encuentran todos dentro de un mismo proceso y por lo tanto pueden compartir los datos globales. Además, una petición bloqueante de un hilo no parará la ejecución de otro hilo. Por último, si el procesador lo soporta, los diferentes hilos están asociados a diferentes conjuntos de registros, por lo que el cambio de contexto del procesador podrá realizase de forma eficiente.

*Multi-threading por Software*

El multi-threading por software posibilita la realización de aplicaciones paralelas. Es un nuevo modelo de programación que permite a múltiples hilos existir dentro de un proceso. Los hilos comparten los recursos del proceso pero se ejecutan de forma independiente. El hecho de que sean independientes permite la concurrencia (es decir su ejecución simultanea), y si el procesador lo soporta se podrán ejecutar en paralelo.

*Multithreading por Hardware*

El multi-threading por hardware es una técnica que incrementa la utilización de los recursos del procesador. A continuación se analizan diferentes tipos:

- Coarse-grained Multithreading

El procesador ejecuta el hilo de forma habitual y solamente realiza un cambio de contexto cuando ocurre un evento de larga duración (como un fallo de caché). Para que el cambio de contexto sea eficiente es necesario que exista una copia del estado de la arquitectura (PC, registros visibles) para cada hilo. Este método tiene la ventaja de ser sencillo de implementar.

- Fine-grained Multithreading

Se basa en un cambio "rápido" entre hilos, ejecutando en cada ciclo un hilo diferente. Es un mecanismo que tiene como base una planificación de la ejecución de las instrucciones en orden. Con el fin de evitar largas latencias por hilos bloqueados, se ejecutan instrucciones de diferentes hilos. Este enfoque tiene la ventaja de eliminar las dependencias de datos que paran el procesador. Al pertenecer las instrucciones a diferentes hilos, las dependencias de datos y de control desaparecen.

- Simultaneous Multithreading

Consiste en poder ejecutar instrucciones de diferentes hilos, en cualquier momento y en cualquier unidad de ejecución. Desarrollar esta tecnología requiere un hardware adicional para toda la lógica. Como consecuencia, su realización para un gran número de hilos aumentaría la complejidad y, por tanto el coste. Por este motivo en las implementaciones SMT se opta por reducir el número de hilos.

SMT muestra a un procesador físico como dos o más procesadores lógicos. Los recursos físicos son compartidos y el estado de la arquitectura es copiada para cada uno de los dos procesadores lógicos. El estado de la arquitectura está formado por un conjunto de registros: registros de propósito general, registros de control, registros del controlador de interrupciones y registros de estado.

Los programas verán a los procesadores lógicos como si se tratara de dos o más procesadores físicos diferentes. Sin embargo, desde el punto de vista de la microarquitectura, las instrucciones de los procesadores lógicos se ejecutarán simultáneamente compartiendo los recursos físicos.

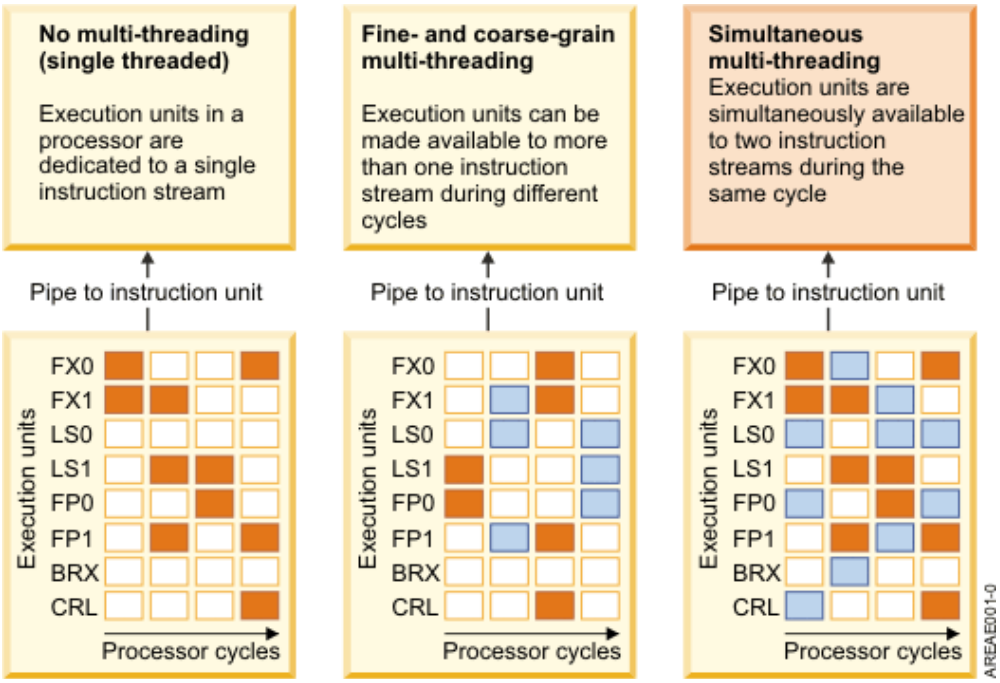


***Fig. 2.5:*** *Comparativa entre procesador single-thread y hardware multi-thread*

### 2.1.6 Procesadores Multicore

Los procesadores multi-core combinan dos o más procesadores (a los que nos referiremos como núcleos o cores) en un mismo chip. Estos procesadores mejoran el rendimiento de las aplicaciones paralelas.

Las aplicaciones paralelas están compuestas por múltiples threads independientes, de forma que es posible la concurrencia. Es decir, los threads se pueden ejecutar al mismo tiempo y en paralelo. Como consecuencia el rendimiento de las aplicaciones paralelas puede teóricamente escalar linealmente con el número de procesadores.

En la práctica existen factores que lo impiden, como los overheads por creación/eliminación de threads, las comunicaciones entre las memorias de los procesadores, y el posible desbalanceo (en las aplicaciones) de volumen de cómputo por thread (threads esperando a que otros threads finalicen). Describiremos estos factores con más detalle, más adelante.

*Arquitectura*

Los procesadores de una arquitectura multi-core comparten la memoria principal. Existen dos alternativas para esta compartición:

- UMA (Acceso uniforme a memoria): los procesadores del sistema tienen el mismo tiempo de acceso a memoria.

- NUMA (Acceso no uniforme a memoria): el acceso a la memoria es controlado por un único procesador, lo que provoca que este procesador tenga un tiempo de acceso menor, a la memoria controlada por él, que el resto de procesadores. El resto de procesadores debe interactuar con el procesador que controla la memoria para acceder a ella.

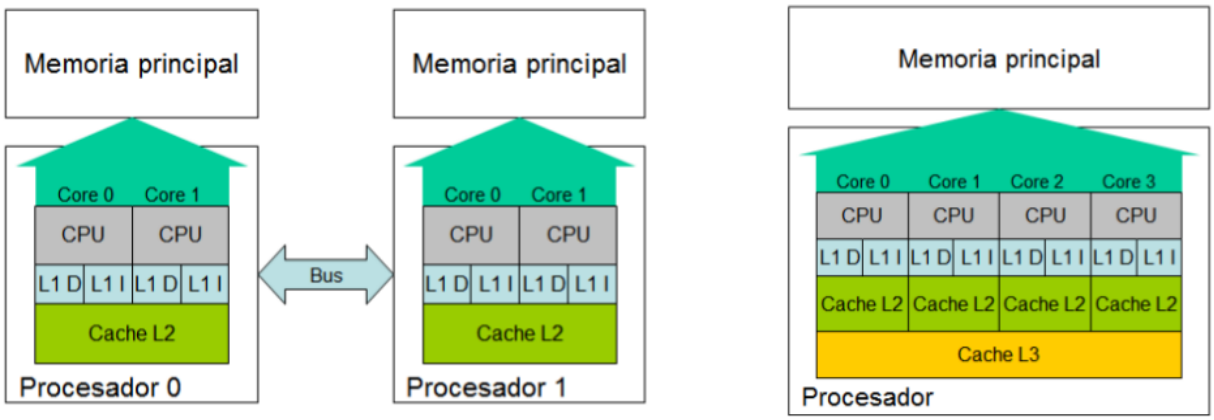


***Fig. 2.6:*** *Comparativa de una arquitectura NUMA (izquierda) y una arquitectura UMA (derecha)*

En cuanto a la organización de las caches de un sistema multi-core hay varias opciones. En algunas arquitecturas se opta por mantener todos los niveles de cache privados a cada core, mientras que en otras arquitecturas se comparte el último nivel de cache.

### 2.1.7 Factores en el rendimiento multi-core/multi-thread

Como hemos visto hasta el momento podría parecer que el aumento paulatino del número de núcleos de un procesador parece la solución a la necesidad de aumentar la capacidad de procesamiento en los procesadores, pero no es así. El rendimiento de las aplicaciones en entornos multi-núcleo y multi-hilo no escala linealmente, y al aumentar el número de hilos con los que se ejecuta la aplicación no se reduce linealmente el tiempo de cómputo de manera transparente al programador. A continuación enunciaremos los principales factores que impiden esta escalabilidad lineal en los entornos de programación multi-núcleo y multi-hilo:

- *Overheads por creación/eliminación de hilos*

Uno de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo, es el proceso de creación y eliminación de hilos (*fork* y *join*) que trabajan en dicho sistema paralelo, que supone un coste de tiempo extra (*overhead*) sobre el total del tiempo de ejecución. Este *overhead* ha de ser mucho menor al tiempo total de ejecución para que sea práctico paralelizar una aplicación.

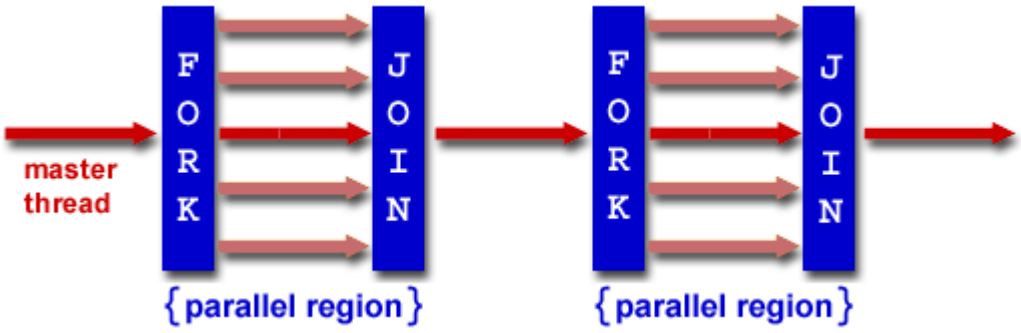


***Fig. 2.7:*** *El proceso de creación y eliminación de threads supone un coste extra de tiempo al total de la ejecución.*

- *Desbalanceo de cómputo*

Otro de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo es la incorrecta distribución del volumen de cómputo por hilo, que implica que algunos hilos finalicen sus tareas antes que el resto y por tanto tendrán que esperar a que el resto acabe para proseguir la ejecución.

Esta espera supone un coste en ciclos de reloj del procesador desaprovechados y por lo tanto un *overhead*. En el diseño de aplicaciones paralelas es muy importante una óptima asignación del trabajo a realizar a cada uno de los hilos. A la hora de programar una aplicación paralela uno de los puntos a tener en cuenta es como de bien se reparte el cómputo entre los diferentes hilos (*computation load balance*).

Este *overhead* se puede reducir aunque no eliminar totalmente, asignando el trabajo dinámicamente entre los diferentes hilos que participan en la ejecución. Por contrapartida se genera otro pequeño *overhead* asociado al cómputo necesario para gestionar la asignación dinámica.

- *Las comunicaciones entre hilos de ejecución*

Otro de los factores que impiden la escalabilidad lineal en sistemas multi-núcleo/multi-hilo es la comunicación entre los hilos. En una región paralela los hilos de una ejecución multi-hilo trabajan de manera independiente y con datos independientes, pero por las características de las aplicaciones, en algún momento necesitarán intercambiar estos datos entre ellos. Este intercambio de datos se realiza de manera transparente al hilo ya que éste únicamente accederá a unas posiciones de memoria que previamente otro hilo habrá modificado. Esto aunque es transparente para el hilo no esta libre de coste en tiempo. Los datos que hayan sido modificados en la cache de un hilo tendrán que ser copiados a la cache del hilo que los necesita en ese momento.

Hay que tener en cuenta que el coste de comunicar datos modificados por hilos que se ejecutan dentro de un mismo procesador es muy inferior al coste de comunicar datos entre hilos que se ejecutan en núcleos de diferentes procesadores.

### 2.1.8 CUDA (*Compute Unified Device Architecture)*

Se trata de una plataforma software que intenta explotar las ventajas de las GPUs de propósito general utilizando el paralelismo que ofrecen sus múltiples núcleos, que permiten el lanzamiento de un altísimo número de hilos simultáneos.

Por ello, si una aplicación está diseñada utilizando numerosos hilos que realizan tareas independientes, una GPU puede ayudar a la CPU en la ejecución de funciones específicas ofreciendo un rendimiento mayor. Así que se podría usar una GPU para la ejecución de algunas funciones de BLAST, por ejemplo en la comparación y extensión de las semillas de las secuencias.

Seria necesario que el sistema usado disponga de una tarjeta gráfica Nvidia compatible. Son compatibles con CUDA todas las GPUs Nvidia de la series GeForce 8000, Quadro y Tesla.

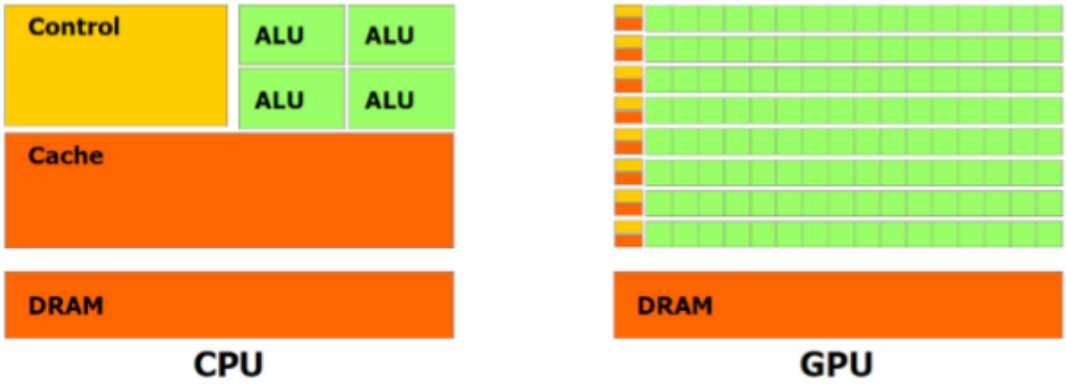


***Fig. 2.8:*** *Esquema de la arquitectura CPU y GPU*

La tecnología CUDA invoca el trabajo a realizar mediante *kernels*. Los threads se organizan en bloques (*blocks*), los bloques están organizados en mallas (*grids*) y cada *grid* solo puede ejecutar un *kernel*.

No todo algoritmo puede ser implementado con CUDA, debe hacer frente a las siguientes limitaciones:

- No se puede utilizar recursividad, punteros a funciones, variables estáticas dentro de funciones o funciones con número de parámetros variable.

- Puede existir un cuello de botella entre la CPU y la GPU por los anchos de banda de los buses y sus latencias.

- Los threads, por razones de eficiencia, deben lanzarse en grupos de al menos 32, con miles de hilos en total.

## 2.2 Bioinformática

La bioinformática es la aplicación de tecnología informática en la gestión y análisis de datos biológicos. Su finalidad puede ser muy variada pero los principales esfuerzos se centran en:

- El alineamiento de secuencias
- La predicción de genes
- El montaje del genoma
- La predicción de la estructura de proteínas
- El alineamiento estructural de proteínas
- La predicción de la expresión génica
- Las interacciones proteína-proteína
- El modelado de la evolución.

### 2.2.1 Fundamentos teóricos de las proteínas

Las proteínas son macromoléculas biológicas que se componen de aminoácidos de los cuales hay veinte distintos. Desempeñan un papel fundamental para la vida y son las biomoléculas más versátiles y más diversas. Son imprescindibles para el crecimiento del organismo. Realizan una enorme cantidad de funciones diferentes, entre las que destacan: estructurales, inmunológicas, transportadoras, protectoras, etc.

La estructura principal de la proteína es la de una secuencia lineal de aminoácidos. Sin embargo, los aminoácidos que forman una proteína interactúan para producir estructuras más complejas. La estructura secundaria se refiere a la disposición espacial de los aminoácidos que se encuentran cerca unos de otros en la secuencia lineal. Algunas de estas estructuras contienen subestructuras, como hélices y hojas. La estructura terciaria se refiere a la disposición espacial de los residuos de aminoácidos que están alejados en la secuencia lineal, que es la conformación tridimensional de la proteína en toda su longitud, que incluye las regiones de estructura secundaria. Otro nivel de la estructura es la estructura cuaternaria que se refiere a la disposición espacial de dos o más proteínas que interactúan (véase figura 2.9).

La estructura terciaria de una proteína implica la funcionalidad de la proteína. Es la interacción entre los aminoácidos de la proteína tanto a nivel local y global la que proporciona su integridad estructural necesaria para llevar a cabo su función biológica. La información para especificar la compleja estructura tridimensional de una proteína está contenida en su secuencia de aminoácidos.

Dos proteínas son homólogas si comparten un ancestro común, es decir, están relacionadas en un contexto evolutivo. Las proteínas homólogas siempre comparten una estructura plegable común de tres dimensiones.

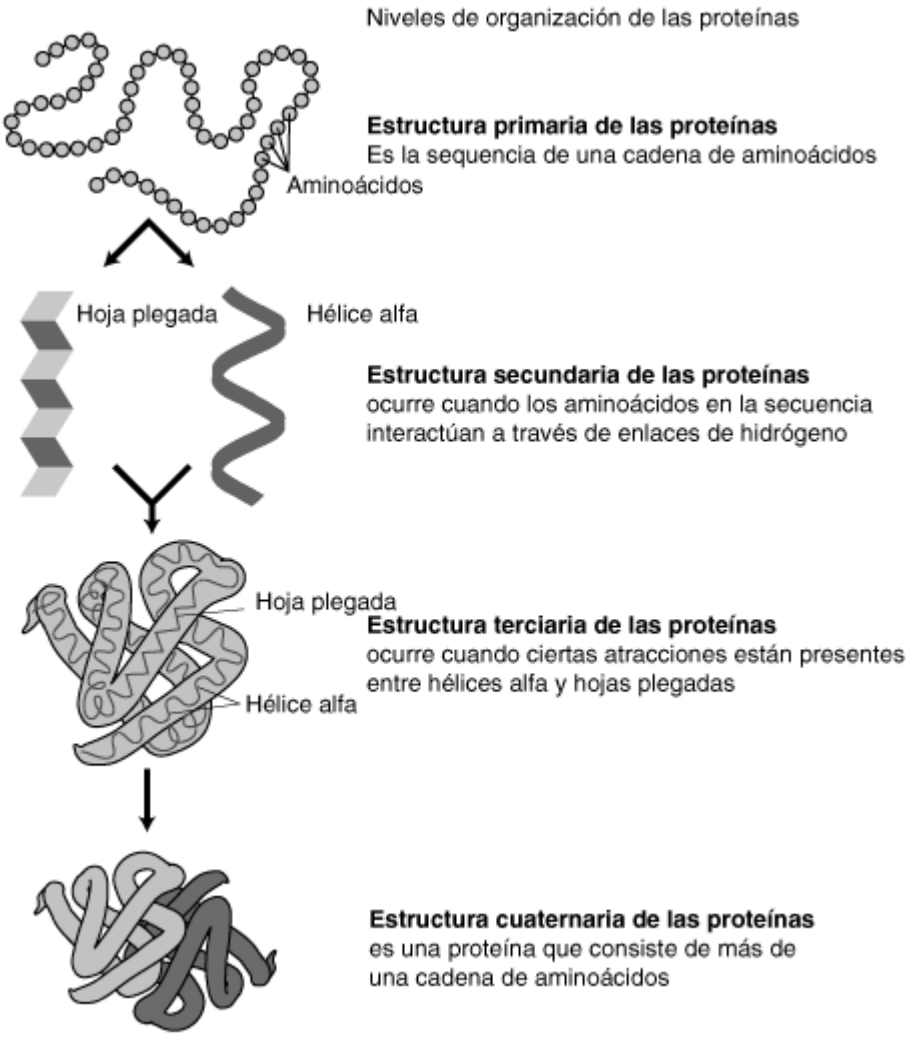


***Fig. 2.9:*** *Estructura de las proteínas*

### 2.2.2 Búsqueda de homología en proteínas

La comparación de una proteína con otra es una de las actividades de cómputo más importantes en la bioinformática. La comparación de secuencias se utiliza como un método para inferir homología y es más informativa cuando detecta proteínas homólogas. La información sobre la estructura secundaria y terciaria de una proteína de la que sólo se conoce la secuencia principal se puede deducir encontrando un homólogo del que se conozca esa información. Dos secuencias cuyas

estructuras principales son similares dentro de un cierto nivel se puede inferir que son homologas. Esta inferencia puede ser posteriormente reforzada por técnicas de comparación que van más allá de la estructura primaria y así ofrecer medidas de similitud basadas en la estructura secundaria y terciaria.

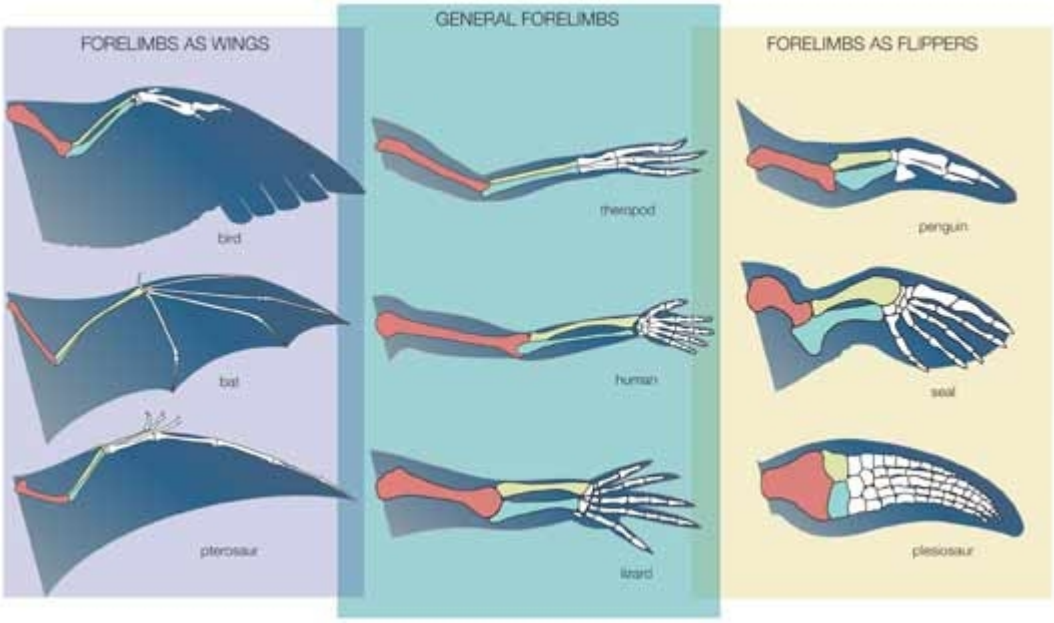


***Fig. 2.10:*** *Ejemplo de homología de las extremidades anteriores. La homología es una característica común entre las especies, que también estuvo presente en su antepasado común.*

La forma de representar las secuencias de proteinas es mediante una cadena de caracteres, donde cada letra representa un aminoácido. Existen 20 aminoácidos distintos y cada uno de ellos tiene asociado un solo carácter. Aunque existen caracteres especiales para representar aminoácidos desconocidos o un grupo de aminoácidos que son casi idénticos entre si.

Por ejemplo, la proteína *B3HSG4 ECOLX* perteneciente a la bacteria *Escherichia coli* se representa con la siguiente secuencia de 39 aminoácidos:

*"MRFTLPGGTAIPEMIDIDHISAFKLLTFLFHPMKLFIFK"*

El proceso de obtener homología consta de varias etapas sucesivas. En primer lugar la representación de una proteína desconocida se compara con las bases de datos de secuencias para encontrar un conjunto de secuencias similares. BLAST o FastA son las herramientas apropiadas para este paso, ya que realizan búsquedas rápidamente con un compromiso aceptable de sensibilidad, es decir, con una capacidad aceptable para evitar falsos positivos.

Siempre que se usa un algoritmo de búsqueda de homología con un par de secuencias se obtiene un alineamiento, incluso aunque las secuencias estén compuestas por letras al azar y no se parezcan nada entre sí. Por lo tanto además hay que estimar la significación estadística de cada homología. Por esta razón tanto BLAST como FastA proporcionan una medida estadística de la significación de cada coincidencia denominada *e-value*, que nos proporciona información sobre si el alineamiento de estas dos secuencias es real o creada por azar.

El uso de las herramientas como BLAST es el primer paso de este proceso en el que una proteína desconocida se compara con una base de datos de secuencias conocidas. Los problemas en este paso son: (1) la elección de un método de puntuación (2) la elección de una base de datos de secuencia de búsqueda (3) la elección de un algoritmo de búsqueda y (4) la evaluación de la significación de los resultados (*e-value*).

Los biólogos moleculares suelen pensar en la homología de aminoácidos en términos de similitud química. La Figura 2.11 muestra una breve clasificación química de los aminoácidos. Desde un punto de vista evolutivo no se espera que las mutaciones cambien radicalmente las propiedades químicas de las proteínas, ya que pueden llegar a destruir sus estructuras tridimensionales. En cambio, las mutaciones entre aminoácidos similares debería ocurrir con relativa frecuencia.

En los años 60 y principios de los 70, Margaret Dayholl fue pionera en técnicas cuantitativas para medir la similitud de aminoácidos. Usando las secuencias de las que se disponía en ese momento, construyó alineamientos múltiples de proteínas relacionadas y comparó las frecuencias de sustituciones de aminoácidos. Como era de esperar, se encontraron pocas variaciones en la frecuencia de sustitución de aminoácidos, y los patrones son generalmente los esperados según las propiedades químicas.

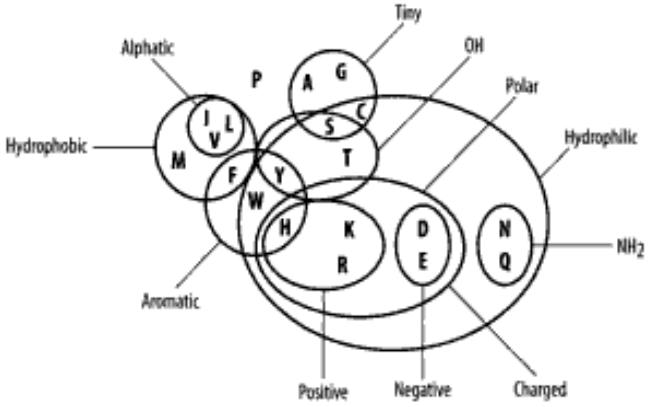


***Fig. 2.11:*** *Relación química entre los aminoácidos*

Por ejemplo, la fenilalanina (F) se encuentra con relativa frecuencia sustituida por la tirosina (Y) y triptófano (W), que comparten las estructuras del anillo aromático (véase la figura 2.11). Y en menor medida con los ácidos hidrofóbicos (V, I y L).

Gracias al estudio de estos patrones que siguen las mutaciones en las proteínas se puede determinar la probabilidad de que una sustitución entre dos aminoácidos ocurra. Hoy en día se usan las matrices de puntuación para estudiar posibles mutaciones proteínicas.

*Matrices de puntuación*

Son una matriz bidimensional que contienen puntuaciones que representan las tasas relativas a sustituciones evolutivas de la proteína, es decir, describe el ritmo al que un aminoácido en una secuencia cambia a otro aminoácido con el tiempo. Las matrices de puntuación son la evolución en pocas palabras. Si observamos las figuras 2.11 y 2.12 podemos observar que las puntuaciones y las propiedades químicas de los aminoácidos a sustituir mantienen cierta relación.

| | C | S | T | P | A | G | N | D | E | Q | H | R | K | M | I | L | V | F | Y | W | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C | 9 | | | | | | | | | | | | | | | | | | | | C |
| S | -1 | 4 | | | | | | | | | | | | | | | | | | | S |
| T | -1 | 1 | 5 | | | | | | | | | | | | | | | | | | T |
| P | -3 | -1 | -1 | 7 | | | | | | | | | | | | | | | | | P |
| A | 0 | 1 | 0 | -1 | 4 | | | | | | | | | | | | | | | | A |
| G | -3 | 0 | -2 | -2 | 0 | 6 | | | | | | | | | | | | | | | G |
| N | -3 | 1 | 0 | -2 | -2 | 0 | 6 | | | | | | | | | | | | | | N |
| D | -3 | 0 | -1 | -1 | -2 | -1 | 1 | 6 | | | | | | | | | | | | | D |
| E | -4 | 0 | -1 | -1 | -1 | -2 | 0 | 2 | 5 | | | | | | | | | | | | E |
| Q | -3 | 0 | -1 | -1 | -1 | -2 | 0 | 0 | 2 | 5 | | | | | | | | | | | Q |
| H | -3 | -1 | -2 | -2 | -2 | -2 | 1 | -1 | 0 | 0 | 8 | | | | | | | | | | H |
| R | -3 | -1 | -1 | -2 | -1 | -2 | 0 | -2 | 0 | 1 | 0 | 5 | | | | | | | | | R |
| K | -3 | 0 | -1 | -1 | -1 | -2 | 0 | -1 | 1 | 1 | -1 | 2 | 5 | | | | | | | | K |
| M | -1 | -1 | -1 | -2 | -1 | -3 | -2 | -3 | -2 | 0 | -2 | -1 | -1 | 5 | | | | | | | M |
| I | -1 | -2 | -1 | -3 | -1 | -4 | -3 | -3 | -3 | -3 | -3 | -3 | -3 | 1 | 4 | | | | | | I |
| L | -1 | -2 | -1 | -3 | -1 | -4 | -3 | -4 | -3 | -2 | -3 | -2 | -2 | 2 | 2 | 4 | | | | | L |
| V | -1 | -2 | 0 | -2 | 0 | -3 | -3 | -3 | -2 | -2 | -3 | -3 | -2 | 1 | 3 | 1 | 4 | | | | V |
| F | -2 | -2 | -2 | -4 | -2 | -3 | -3 | -3 | -3 | -3 | -1 | -3 | -3 | 0 | 0 | 0 | -1 | 6 | | | F |
| Y | -2 | -2 | -2 | -3 | -2 | -3 | -2 | -3 | -2 | -1 | 2 | -2 | -2 | -1 | -1 | -1 | -1 | 3 | 7 | | Y |
| W | -2 | -3 | -2 | -4 | -3 | -2 | -4 | -4 | -3 | -2 | -2 | -3 | -3 | -1 | -3 | -2 | -3 | 1 | 2 | 11 | W |

***Fig. 2.12:*** *Matriz de puntuación BLOSUM62*

Para mayor precisión los resultados se multiplican por un factor de escala antes de convertirlos en enteros. Por ejemplo, una puntuación de -1.609 *nats* (medida usada para estas puntuaciones) puede ser escalado por un factor de dos y luego redondeado a un valor entero de -3. Las puntuaciones que se han reducido y escalado a números enteros tienen una cantidad sin unidades y se llaman puntajes brutos.

Existen dos tipos de matrices de puntuación: PAM (*Point-Accepted Mutation*) y BLOSUM (*BLOcks of Amino Acid SUbstitution Matrix*). Las matrices PAM fueron creadas por el equipo científico de Margaret Dayhoff, por lo que a veces se llaman matrices Dayhoff. Estas matrices de puntuación tienen un componente teórico fuerte y hacen uso de algunas hipótesis evolutivas. En cambio, las matrices BLOSUM son más empíricas y derivan de un conjunto de datos mayor. La mayoría de los investigadores prefieren utilizar matrices BLOSUM ya que en los experimentos demuestran mayor sensibilidad.

*Bases de datos (databases)*

Las bases de datos contienen una colección de todas las secuencias conocidas que son públicamente accesibles. Cada una de las entradas de estas bases de datos es una secuencia, de ADN o proteínas, que contiene una serie de anotaciones en las que se especifican sus características.

Para una utilización más racional de toda la información almacenada en las bases de datos, las distintas secuencias se han agrupado en diferentes categorías que se denominan divisiones, algunas de las cuales reflejan un origen filogenético y otras se basan en la aproximación técnica que se usó para generarlas (por ejemplo: hongos, bacterias, mamíferos, virus, etc.). Independientemente de las divisiones en que se agrupen las secuencias, todas ellas se obtienen de los datos enviados por los investigadores, que voluntariamente hacen accesibles sus datos a toda la comunidad científica.

Además de la secuencia en sí, en las *databases* se almacena mucha más información para cada secuencia. Por ejemplo, en la *database* de proteínas Swiss-Prot se dispone de los siguientes campos para cada secuencia conocida (figura 2.13):

| Line code | Content |
|---|---|
| ID | Identification |
| AC | Accession number(s) |
| DT | Date |
| DE | Description |
| GN | Gene name(s) |
| OS | Organism species |
| OG | Organelle |
| OC | Organism classification |
| OX | Taxonomy cross-reference(s) |
| RN | Reference number |
| RP | Reference position |

| Line code | Content |
|---|---|
| RC | Reference comment(s) |
| RX | Reference cross-reference(s) |
| RA | Reference authors |
| RT | Reference title |
| RL | Reference location |
| CC | Comments or notes |
| DR | Database cross-references |
| KW | Keywords |
| FT | Feature table data |
| SQ | Sequence header |
| (no code) | (blanks) sequence data |

***Fig. 2.13:*** *Registros en la database Swiss-Prot*

Aunque desde el punto de vista histórico y científico las bases de datos tradicionalmente se han separado en secuencias de ácidos nucleicos y de proteínas, mantenidas de forma independiente y con poca relación entre unas y otras, en la actualidad se está imponiendo la tendencia contraria, es decir, una mayor relación entre ambas que facilite el obtener toda la información disponible lo más fácilmente posible.

Las bases de datos crecen a un ritmo exponencial: la *database* Swiss-Prot se duplica cada 40 meses y las bases de datos de ADN lo hacen cada 14 meses. Esta tendencia se ha visto incrementada desde que las secuencias EST comenzaron a añadirse (1994) y se mantendrá probablemente hasta que se completen los proyectos de secuenciación del genoma humano y del ratón. Por entonces las bases de datos tendrán entre 50 y 200 GB de información. Con esta complejidad, las búsquedas en las bases de datos necesitarán computadores de mayor prestaciones, por lo que cada vez son menos las organizaciones que guardan copias locales de las bases de datos.

*Algoritmos de búsqueda de homología*

Los algoritmos de búsqueda de homología se clasifican según el tipo (programación dinámica o heurística) y según la alineación (global o local) (figura 2.14). Los algoritmos de programación dinámica son más costosos computacionalmente, pero son menos propensos a pasar por alto una coincidencia significativa. Son los métodos usados cuando es necesaria una comparación rigurosa.

| Algorithm Type | Alignment Type | |
|---|---|---|
| | Global | Local |
| Dynamic Programming | Needleman-Wunsch | Smith-Waterman |
| Heuristic | FASTA | BLAST |

***Fig. 2.14:*** *Clasificación de algoritmos para la comparación de secuencias de proteínas.*

Los algoritmos heurísticos tienen un coste computacional menor, pero pueden pasar por alto regiones de similitud interesantes, es decir, regiones en las que la medida de similitud supera un valor preestablecido. Son los métodos más usados debido a sus requisitos computacionales relativamente bajos. Pero el tipo de alineamiento que produce es también un factor importante. El alineamiento local es capaz de reconocer mejor posibles regiones de gran similitud entre dos secuencias, mientras que el alineamiento global es perjudicado por los espacios entre estas regiones. Los algoritmos de alineación global son a menudo la elección si dos secuencias son conocidas y a priori relacionadas. Sin embargo, las proteínas lejanamente relacionadas son más propensas a ser similares en las subregiones y es recomendable usar algoritmos de alineación local.

Los algoritmos de programación dinámica no son prácticos para las búsquedas en bases de datos por su elevado coste en tiempo. Por lo que se apuesta por un desarrollo de algoritmos heurísticos que sacrifican la sensibilidad de los resultados por la velocidad. El más popular de estos son FastA, para el cálculo de alineamientos globales, y BLAST, para el cálculo de alineamientos locales.

## 2.3 BLAST

BLAST (Basic Local Alignment Search Tool) es un software libre de búsqueda de similitud de secuencias que puede ser usado para buscar una secuencia desconocida (*query*) en una base de datos de secuencias (*database*). Se trata de uno de los instrumentos bioinformáticos más populares.

El conjunto de aplicaciones BLAST está desarrollado por el NCBI (*National Center for Biotechnology Information*). La versión actual está diseñada en C++, pero al estar creada a partir de la versión escrita en C del año 1997 las aplicaciones BLAST están compuestas tanto por código C como C++.

A pesar de que BLAST es un programa muy poderoso y casi siempre podemos confiar en sus resultados, puede que no encuentre la solución óptima ya que se trata de un programa heurístico. No garantiza que las secuencias que alinea sean homólogas y mucho menos que tengan la misma función, simplemente provee posibles candidatos. Se necesitan más análisis para anotar correctamente una secuencia.

### 2.3.1 Versiones y variantes de BLAST

BLAST contiene muchas versiones distintas ya que las secuencias con las que trabaja pueden ser de distintos tipos: nucleótidos (ADN), proteínas, de ADN traducidas a proteínas o a la inversa.

- Blastn: es de los más comúnmente usados. Compara una secuencia de nucleótidos (ADN) contra una base de datos que contenga también secuencias nucleotídicas.

- Blastp: es el otro tipo de BLAST más usado. Es un BLAST con huecos (o *gaps*) que compara una secuencia de aminoácidos contra una base de datos del mismo tipo.

- BlastX: este programa usa como entrada una secuencia de nucléotidos. Traduce la secuencia y la compara contra una base de datos de proteínas. Se usa cuando se tiene sospecha de que la secuencia de entrada codifica para una proteína.

- Tblastn: compara una proteína con una base de datos de nucléotidos. Se usa cuando el análisis con *Blastp* no ha sido exitoso con dicha proteína.

- TblastX: es la combinación del *TBlastn* y *BlastX*. Compara una secuencia de nucleótidos contra una base de datos de nucleótidos, pero primero traduce tanto la secuencia problema como la base de datos a proteínas.

- Bl2seq: compara dos secuencias entre ellas, en vez de comparar una secuencia con una base de datos.

Además, se pueden encontrar diferentes variantes de BLAST:

- Gapped Blast o BLAST 2.0: esta es una mejora al algoritmo original del BLAST. Actualmente es la forma usual de BLAST que se usa. Se trata de un BLAST que contempla la existencia de pequeñas inserciones o eliminaciones en las secuencias que se están comparando, permitiendo así alinear uno o varios nucléotidos o aminoácidos con huecos vacíos llamados *gaps*. El uso de este nuevo enfoque, agrega dos parámetros al algoritmo, uno es la penalización que se da en la puntuación por alinear un nucleótido o aminoácido con un *gap* y el otro es una penalización por extender un *gap* preexistente. Siempre se considera más costoso abrir un nuevo *gap* que expandir uno existente.

- PsiBlast: esta variante de BLAST se usa para buscar posibles homólogos en organismos muy lejanos entre ellos, filogenéticamente hablando. Está disponible sólo para secuencias de aminoácidos. Se trata de un programa iterativo que va calculando su propia matriz de sustitución en cada iteración. Al inicio, hace un *Blastp* normal, usando una matriz estándar para calificar los alineamientos. De las secuencias obtenidas en este alineamiento, el programa genera una nueva matriz de sustitución, basándose en los alineamientos obtenidos. Usa esta nueva matriz para realizar otro alineamiento. Esto permite en general encontrar nuevos alineamientos, que son usados para calcular una nueva matriz. El proceso se repite tantas veces como el usuario lo indique, o hasta que ya no se encuentran nuevos alineamientos.

- WUBlast: es el algoritmo de BLAST implementado por bioinformáticos de la Universidad de Washington. Según sus creadores, es un algoritmo mucho más rápido y eficiente que el BLAST de NCBI, e igual de sensible. Es ideal si se quieren realizar análisis masivos de BLAST. Otra diferencia es la licencia, WU BLAST es software propietario y es gratuito solo para uso académico.

### 2.3.2 Input

Para la ejecución de BLAST solo se necesitan dos parámetros de entrada obligatorios:

- *Query o queries:* secuencias desconocidas sobre las que se va a realizar la búsqueda de similitud. Deben estar escritas en un fichero en formato FASTA. Este formato basado en texto es utilizado para representar secuencias bien de ácidos nucleicos o proteínas, y en el que los pares de bases o aminoácidos se representan usando códigos de una única letra. El formato también permite incluir nombres de secuencias y comentarios que preceden a las

secuencias en sí. La simplicidad del formato FASTA hace fácil el manipular y analizar secuencias usando herramientas de procesado de textos y scripts.

```
>gi|5524211|gb|AAD44166.1| cytochrome b [Elephas maximus maximus]
LCLYTHIGRNIYYGSYLYSETWNTGIMLLLITMATAFMGYVLPWGQMSFWGATVITNLFSAIPYIGTNLV
EWIWGGFSVDKATLNRFFAFHFILPFTMVALAGVHLTFLHETGSNNPLGLTSDSDKIPFHPYYTIKDFLG
LLILILLLLLLALLSPDMLGDPDNHMPADPLNTPLHIKPEWYFLFAYAILRSVPNKLGGVLALFLSIVIL
GLMPFLHTSKHRSMMLRPLSQALFWTLTMDLLTLTWIGSQPVEYPYTIIGQMASILYFSIILAFLPIAGX
IENY
```

***Fig. 2.15:*** *Ejemplo de secuencia escrita en formato FASTA*

- *Database:* se debe especificar con qué base de datos se quiere realizar la búsqueda. Es importante usar la *database* correcta. No podemos hacer una búsqueda con *Blastp* con una *query* proteínica y una *database* de ADN.

### 2.3.3 Output

La salida de una ejecución de BLAST puede ser por pantalla o por fichero. En ambos casos, todos los programas de BLAST proporcionan información en más o menos el mismo formato. Primero viene una introducción al programa, una serie de descripciones de las secuencias homólogas encontradas (nombre, *e-value*, *score*, etc.), las alineaciones de estas secuencias y, finalmente, los parámetros y otros datos estadísticos recopilados durante la búsqueda.

- Introducción del programa: proporciona el nombre del programa, el número de versión, la fecha en que el código fuente fue cambiado sustancialmente, la fecha en que se construyó el programa y una descripción de la secuencia de consulta y base de datos que desea buscar (figura 2.16).

```
BLASTP 2.2.24+

Database: All non-redundant GenBank CDS translations+PDB+SwissProt+PIR+PRF
excluding environmental samples from WGS projects
           12,061,831 sequences; 4,118,133,053 total letters


Query=  AATM_RABIT P12345 Aspartate aminotransferase, mitochondrial (EC
2.6.1.1) (Transaminase A) (Glutamate oxaloacetate transaminase-2)
(Fragment).
Length=30
```

***Fig. 2.16:*** *Fragmento inicial del output de Blastp*

- Secuencias homólogas: son las descripciones de las secuencias obtenidas (en una línea). A menos que se solicite lo contrario, las secuencias homólogas se ordenan mediante el valor de probabilidad *e-value*, es decir, se ordenan de mayor a menor significación (figura 2.17).

```
Sequences producing significant alignments:                          (Bits)  Value

ref|XP_001495474.1|  PREDICTED: similar to Aspartate aminotransfe...  68.6    2e-10
ref|XP_002711597.1|  PREDICTED: Aspartate aminotransferase, mitoc...  68.6    2e-10
sp|P08907.1|AATM_HORSE  RecName: Full=Aspartate aminotransferase,...  68.6    3e-10
prf||1003180A  aminotransferase,Asp                                   68.6    3e-10
ref|NP_001016933.1|  aspartate aminotransferase, mitochondrial pr...  67.8    4e-10
prf||0308236A  aminotransferase,Asp                                   67.0    6e-10
prf||0410468A  aminotransferase,Asp                                   66.2    1e-09
pdb|1AKA|A  Chain A, Structural Basis For The Catalytic Activity ...  66.2    1e-09
pdb|7AAT|A  Chain A, X-Ray Structure Refinement And Comparison Of...  66.2    1e-09
ref|NP_001080255.1|  aspartate aminotransferase 2 [Xenopus laevis...  65.9    1e-09
ref|XP_002187671.1|  PREDICTED: similar to aspartate aminotransfe...  65.9    2e-09
sp|P12345.1|AATM_RABIT  RecName: Full=Aspartate aminotransferase,...  65.9    2e-09
```

***Fig. 2.17:*** *Fragmento del listado de secuencias obtenidas (ordenadas por probabilidad)*

- Alineaciones: Alineaciones de las secuencias homólogas encontradas (figura 2.18).

```
>ref|XP_001495474.1| PREDICTED: similar to Aspartate aminotransferase, mitochondrial
precursor (Transaminase A) (Glutamate oxaloacetate transaminase
2) (mAspAT) (Fatty acid-binding protein) (FABP-1) (FABPpm)
[Equus caballus]
Length=430

 Score = 68.6 bits (166),  Expect = 2e-10, Method: Compositional matrix adjust.
 Identities = 30/30 (100%), Positives = 30/30 (100%), Gaps = 0/30 (0%)

Query  1   SSWWAHVEMGPPDPILGVTEAYKRDTNSKK  30
           SSWWAHVEMGPPDPILGVTEAYKRDTNSKK
Sbjct  30  SSWWAHVEMGPPDPILGVTEAYKRDTNSKK  59
```

***Fig. 2.18:*** *Ejemplo de alineación en el output*

- Parámetros y estadísticas: parámetros y otras estadísticas recogidas durante la búsqueda en la base de datos (figura 2.19).

```
Lambda     K      H
   0.310    0.128    0.427

Gapped
Lambda     K      H
   0.267   0.0410    0.140

Effective search space used: 101445597450

Matrix: BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
Neighboring words threshold: 11
Window for multiple hits: 40
```

***Fig. 2.19:*** *Fragmento sobre parámetros y estadísticas*

### 2.3.4 Algoritmo de BLASTP

*Blastp* es una de las versiones más usadas y más rápidas aunque utilice alineaciones con huecos (*gaps)*, que son más costosas. Compara una o varias secuencias de aminoácidos (proteínas) con secuencias del mismo tipo almacenadas en una base de datos (*database*) para encontrar secuencias homólogas ya conocidas.

Usando un método heurístico, *Blastp* encuentra secuencias homólogas, pero no mediante la comparación de la secuencia en su totalidad, sino por la localización de pequeñas coincidencias entre las dos secuencias.

Para encontrar estas pequeñas coincidencias se hace un proceso inicial llamado *seeding* que realiza una búsqueda de palabras o *words*, que son tripletas de aminoácidos, en la secuencia de interés (*query*). Es decir, la secuencia de aminoácidos de entrada se divide en tripletas y se almacenan.

Esta lista de tripletas almacenadas son utilizadas para construir una alineación con las tripletas resultantes de las secuencias conocidas (*subjects*).

Utilizando una matriz de puntuación, cada alineación debe obtener una puntuación superior a un umbral T para ser considerada como significativa. Una vez las alineaciones han sido evaluadas se extienden en ambas direcciones. Cada extensión afecta en la puntuación de la alineación, ya sea aumentando o disminuyendo la misma.

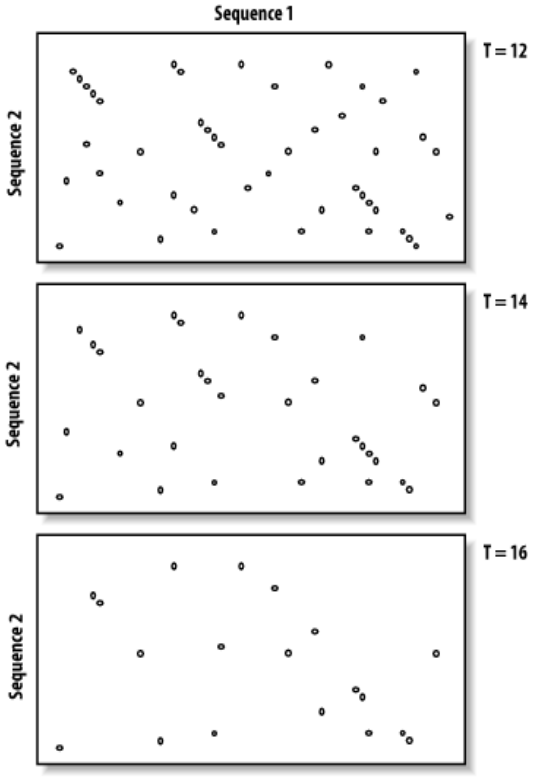


***Fig. 2.20:*** *Efecto del umbral T. El aumento de T acelera el proceso pero disminuye el número de hits.*

Si esta puntuación es superior a otro umbral T' determinado, la alineación se incluirá en los resultados dados por *Blastp*. Sin embargo, si esta puntuación es inferior a ese umbral, la extensión deja de aplicarse, evitando que se incluyan en los resultados áreas de una mala alineación.

El aumento del umbral T acelera el proceso de *Blastp*, pero también limita la búsqueda disminuyendo el número de palabras coincidentes (figura 2.20).

De esta forma se puede dividir *Blastp* en tres módulos o fases (Figura 2.21). La fase de setup establece la búsqueda. La fase de escaneo analiza cada secuencia conocida en busca de palabras coincidentes que después extiende. Y por último, la fase de rastreo que produce una alineación total de la secuencia con *gaps* con inserciones y eliminaciones.

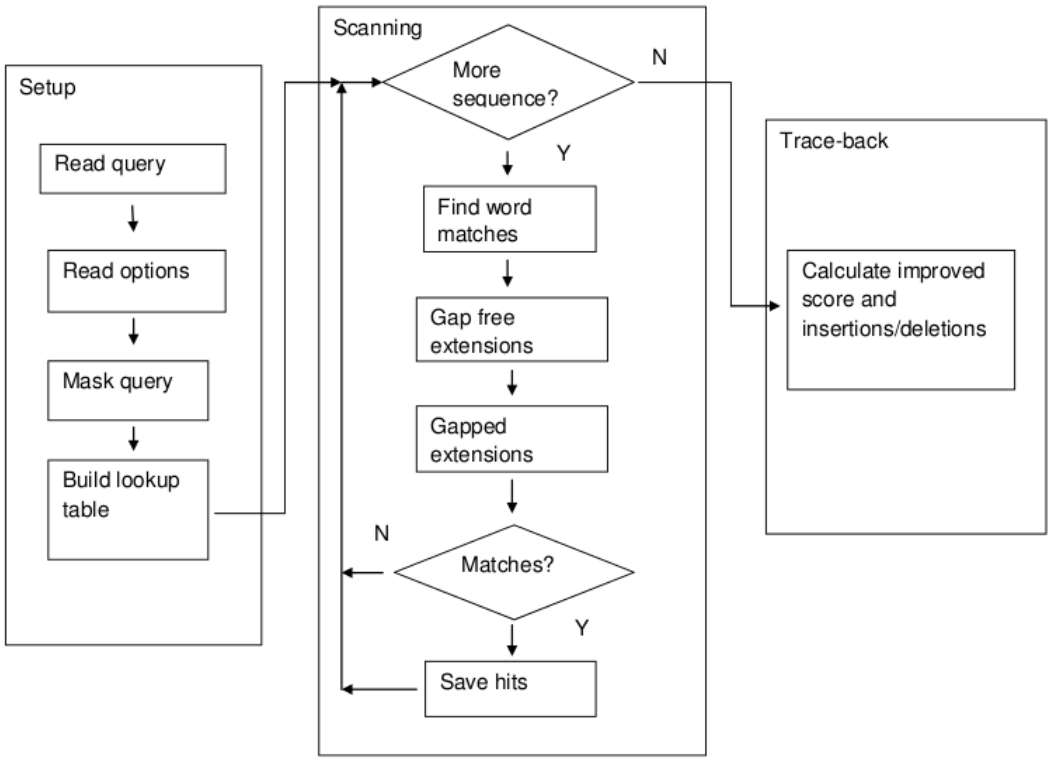


***Fig. 2.21:*** *Diagrama de flujo del algoritmo BLAST*

- Fase 1: SETUP

La fase de setup lee la secuencia de consulta y construye una *lookup table*. La *lookup table* (LUT) es una tabla *hash* que contiene las palabras o *words* de la *query* y su posición o desplazamiento.

Las *words* son las tripletas de aminoácidos de las que se compone la *query.* Es decir, la primera tripleta corresponde a los aminoácidos de la posición 1, 2 y 3, la siguiente a los de la posición 2, 3 y 4, …, hasta llegar a las posiciones n-2, n-1 y n, dónde n corresponde a la longitud de *query.*

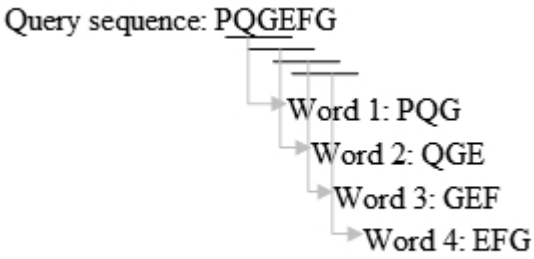


***Fig. 2.22:*** *Ejemplo de construcción de la lista de words*

- Fase 2: SCANNING

La fase de escaneo explora la base de datos y realiza las extensiones. Cada secuencia conocida se analiza en busca de palabras coincidentes (*hits*) con los de la *lookup table* (LUT). Las coincidencias obtenidas se utilizan para iniciar una alineación extendida sin huecos (*gaps)*. El proceso de alineamiento sin huecos (*ungapped alignment*) extiende los *hits* en cada dirección en un intento de incrementar su puntuación de alineación. Para calcular la puntuación de las alineaciones se hace uso de las matrices de puntuación. Las inserciones y eliminaciones no son consideradas durante esta etapa.

En el siguiente ejemplo gráfico (figura 2.23) podemos ver los *hits encontrados* después de analizar dos secuencias. Se puede observar como hay coincidencias agrupadas representando pequeñas diagonales. Estas diagonales son regiones que coinciden entre dos secuencias que la alineación sin *gaps* va a detectar como alineaciones de mayor puntuación y va a intentar extender por ambos lados (figura 2.23).

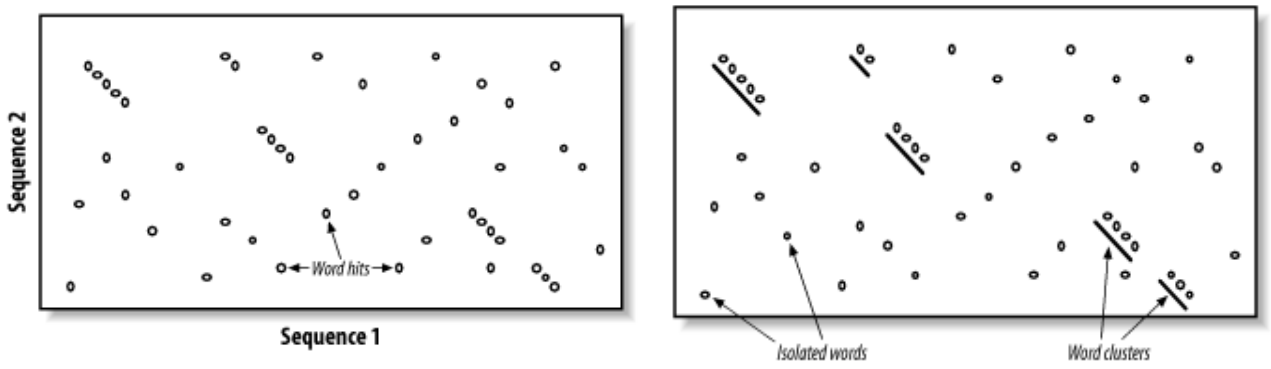


***Fig. 2.23:*** *Ejemplo de hits encontrados al comparar dos secuencias distintas, se pueden observar regiones en común entre las dos secuencias.*

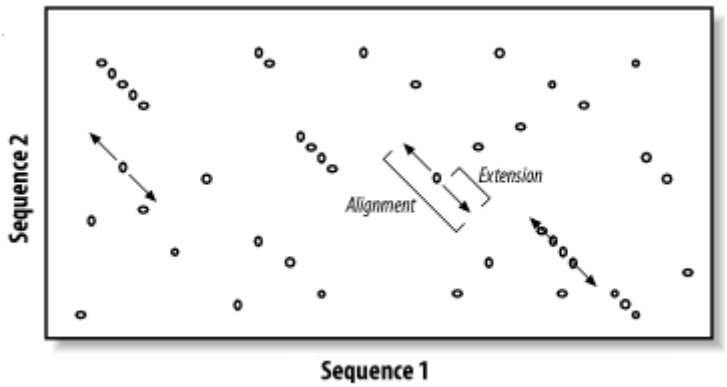


***Fig. 2.24:*** *Ejemplo de extensión de los hits, al extender se intenta incrementar la puntuación de alineación de cada coincidencia encontrando así alineaciones de mayor tamaño.*

Aquellas alineaciones que superen un umbral de puntuación T, inician una alineación extendida con *gaps*. La extensión mediante alineación con *gaps* se aplica mientras la puntuación de la alineación sea superior a otro umbral T'. Cuando deja de superar este nuevo umbral se deja de aplicar la extensión y la alineación se almacena para su posterior procesamiento.

En muchas ocasiones, es necesario introducir huecos (*gaps*) en el alineamiento para compensar las inserciones y eliminaciones que afectan a las secuencias a lo largo de la evolución. Sin embargo, si permitimos la inserción de numerosos *gaps* en el alineamiento, podríamos llegar a alinear dos secuencias completamente divergentes. Para evitar que esto ocurra los programas de alineamiento introducen una penalización en la puntuación del alineamiento por cada hueco que se abre (G o *gap opening penalty*) y otra adicional en función de la longitud del hueco (L o *gap extension penalty*).

Estos dos parámetros pueden ser fijados por el usuario dependiendo de sus intereses, aunque lo normal es utilizar un valor grande de G y bajo para L, asumiendo que en la naturaleza los acontecimientos de inserción/eliminación son raros (G grande), pero una vez que ocurren pueden afectar a varios residuos adyacentes (L pequeña).

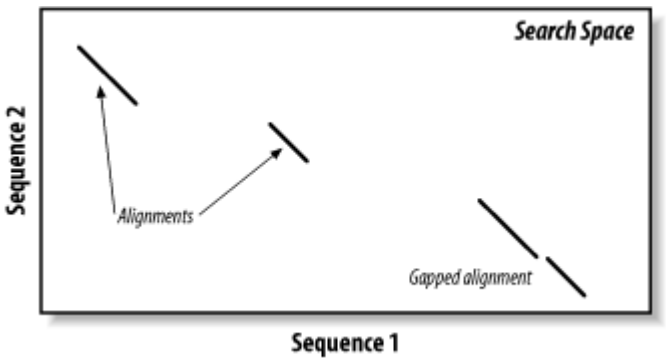


***Fig. 2.25:*** *Ejemplo de alineaciones ungapped y alineación gapped.*

La fase de escaneo cuenta con alguna optimización. La alineación *gapped* (con huecos) devuelve sólo la puntuación y el alcance de la alineación. El número y posición de las inserciones, eliminaciones y los aminoácidos coincidentes no se almacenan, reduciendo el tiempo de CPU y memoria de las ejecuciones.

- Fase 3: TRACE-BACK

La fase final de la búsqueda BLAST es el rastreo o *trace-back*. También se conoce por la fase de evaluación ya que en ella se obtiene la puntuación de la alineación final de la secuencia. Se genera una alineación final con *gaps* usando las alineaciones obtenidas en la fase anterior. Al final sólo se reportan los alineamientos que hayan obtenido una probabilidad mayor a E. El parámetro E es conocido como valor de corte (*e-value*), y nos permite definir qué alineamientos queremos obtener de acuerdo a su significación estadística. Cuanto menor sea el valor de E, más significativo es un alineamiento. En la siguiente figura podemos ver un ejemplo gráfico de esta fase (figura 2.26):

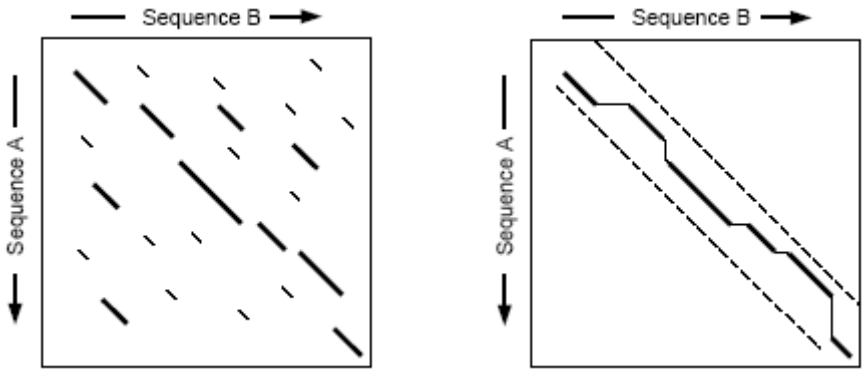


***Fig. 2.26:*** *Ejemplo gráfico sobre el trace-back de Blastp. De las alineaciones resultantes de la fase anterior (fase de escaneo) se produce una alineación con gaps final para obtener la alineación con mejor puntuación.*

### 2.3.5 Lookup Table

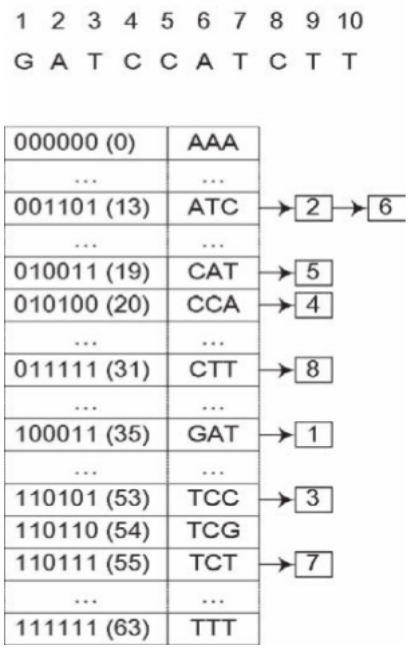


***Fig. 2.27:*** *Ejemplo de lookup table para secuencias de ADN con alfabeto de 4 caracteres (A,C,G,T) y W=3 bases*

El primer paso de *Blastp* es aislar todas las tripletas (W=3 aminoácidos) de la secuencia de consulta y almacenar el desplazamiento en la que cada palabra se produce. Estos desplazamientos representan la tabulación de las *words* en la *query*. Esta información se almacena en una tabla hash llamada *lookup table* (LUT).

Más tarde, con una secuencia conocida de una base de datos (*subject*), el trabajo consiste en deslizarse a través de cada *word* de la *subject* y encontrar *hits*, que consistiría en recuperar de la LUT las palabras que tienen una buena puntuación cuando se alinean con dicha *word*.

El alfabeto de *Blastp* contiene 28 caracteres y las words son de W=3 caracteres. Básicamente, cada una de las posibles *words* se direcciona a una ubicación única en la LUT. La función hash considera cada letra como un número entre 0 y 27 que puede caber en 5 bits, así que para almacenar *words* de W=3 aminoácidos se necesitan 5·3=15 bits, dando un tamaño a la LUT de $2^{15}$=32768 posibles entradas.

Para mejorar el rendimiento de las operaciones de búsqueda, BLAST utiliza un array de bits llamado PV (*presence vector*) que es consultado antes de acceder a la LUT. Este array tiene una longitud igual al número de entradas de la LUT, y nos indica mediante un bit si una entrada está vacía o no. Aprovechando que PV puede permanecer en caché por su pequeño tamaño podemos evitar innecesarias y costosas búsquedas en la LUT cuando una entrada está vacía.

# Capítulo 3

# Análisis de rendimiento

Este capítulo está dedicado al análisis de rendimiento de la aplicación *blastp* descrita en el capítulo anterior. Para encontrar los motivos de un bajo rendimiento en una aplicación es necesario estudiar y entender cómo se estructura y cómo funciona. Por este motivo se ha hecho un estudio para observar el comportamiento de la aplicación en distintos contextos. Hay que fijarse en aquellas funciones llamadas por *blastp* en las que el ordenador dedica más tiempo de ejecución, tanto de cómputo como de acceso a disco o memoria. De esta manera sabremos qué regiones de código se necesita estudiar y de las cuales buscar soluciones para mejorar el rendimiento de la aplicación.

## 3.1 Análisis de los datos de entrada y parámetros

Tanto *blastp* como todas las variantes de BLAST disponen de una multitud de parámetros de entrada, de los cuales sólo se van a comentar los que más efecto tienen en el rendimiento:

- Query o queries: es la secuencia o secuencias problema que queremos comparar con las secuencias conocidas y almacenadas en una base de datos de proteínas.

- Database: especifica qué base de datos usaremos de las distintas de que dispone BLAST.

- Número de threads (*opcional*): especifica el número de threads que queremos lanzar para ejecutar la versión multi-core de la aplicación. En un procesador multi-core podemos aprovechar los diferentes núcleos para ejecutar diferentes threads en paralelo y así obtener un rendimiento mayor.

### 3.1.1 Queries

Como se va a ver en este capítulo, las *queries* tienen una gran repercusión en el rendimiento de la aplicación. Además, se debe tener en cuenta que los resultados del rendimiento son muy distintos cuando solo tenemos una *query* o tenemos varias de ellas.

En concreto hay dos factores en las *queries* que son clave en el rendimiento de *blastp*:

- Longitud de la secuencia query: Cuanto mayor es el tamaño de la secuencia *query*, mayor es la cantidad de instrucciones ejecutadas. Además, el tamaño que ocupará la LUT en memoria depende también del tamaño de las secuencias de entrada. El tamaño de la LUT afecta al rendimiento, ya que si tiene un tamaño excesivo puede hacer aumentar el número de accesos (conflictos en la tabla *hash*) y fallos de caché. Así que la longitud de las *query* es un factor muy importante que debe ser estudiado.

- Información proteica de la *query*: La información de la proteína o los aminoácidos que aparecen en ella son también factores a tener en cuenta. Dependiendo de las *words* o tripletas de aminoácidos por las que está compuesta una proteína, *blastp* tendrá que realizar una carga de trabajo distinta. Es decir, la aparición de determinadas tripletas afectan al rendimiento de la aplicación. Cuanta más diversificación en las *words* más *words* hay que almacenar en la LUT provocando un aumento de su tamaño, y además, si se da el caso que hay *words* muy comunes en la base de datos se tendrán que almacenar más *hits* en memoria.

Cuando solo disponemos de una *query* veremos cómo se comporta *blastp* al variar su longitud. Para esto haremos uso de las siguientes *queries* distintas en tamaño y en información proteica, con tamaños múltiples de 2 entre ellas (Figura 3.1).

| Secuencia | ID | Especie | Longitud |
|---|---|---|---|
| 1 | A4T9V0 | Mycobacterium gilvum | 64 |
| 2 | Q2IJ63 | Anaeromyxobacter dehalog. | 128 |
| 3 | P28484 | Drosophila teissieri | 256 |
| 4 | Q1JLB7 | Streptococcus pyogenes | 512 |
| 5 | P08715 | Escherichia coli | 1024 |
| 6 | Q8IYD8 | Homo sapiens | 2048 |
| 7 | Q06277 | Adenosine monophosphate | 4096 |

***Figura 3.1:*** *Secuencias usadas para una query*

Cuando dispongamos de varias *queries* a la vez usaremos 16 *queries* de distinta información proteica pero de la misma longitud (Figura 3.2). Todas contienen 240 aminoácidos, un tamaño bastante común en las ejecuciones con *blastp*.

| Secuencia | ID | Especie | Longitud |
|---|---|---|---|
| 1 | P46969 | Saccharomyces cerevisiae | 240 |
| 2 | Q9L0Z5 | Streptomyces coelicolor | 240 |
| 3 | Q9CCP9 | Mycobacterium leprae | 240 |
| 4 | P71676 | Mycobacterium tuberculosis | 240 |
| 5 | O34557 | Bacillus subtilis | 240 |
| 6 | P74061 | Synechocystis sp. | 240 |
| 7 | P51012 | Rhodobacter capsulatus | 240 |
| 8 | Q43843 | Solanum tuberosum | 240 |
| 9 | P32661 | Escherichia coli | 240 |
| 10 | Q9ZTP5 | Oryza sativa Japonica | 240 |
| 11 | E1EQP8 | Enterococcus TUSoD | 240 |
| 12 | E1L132 | Atopobium vaginae | 240 |
| 13 | E0FZA3 | Enterococcus TX4248 | 240 |
| 14 | E1YJS2 | Desulfobacterium | 240 |
| 15 | Q8GC94 | Citrobacter freundii | 240 |
| 16 | E1Q8R2 | Helicobacter pylori Cuz20 | 240 |

***Figura 3.2:*** *Secuencias usadas para varias queries*

### 3.1.2 Base de datos (NR)

La base de datos escogida para el análisis de rendimiento es la NR. Se trata de una base de datos (o *database*) que contiene 12.062.381 secuencias de proteínas ocupando unos 9.5 GB de memoria en disco. Se trata de la base de datos disponible por el NCBI más grande y más usada, aunque hay que mencionar que en el contexto científico no siempre la base de datos más grande es la que da mejores resultados, pero para el análisis del rendimiento es interesante ver cómo se comporta *blastp* trabajando con datos de gran volumen. La *database* NR incluye otras bases de datos como la GenBank, RefSeq, EMBL (base de datos europea), DDBJ (base de datos japonesa) y PDB (banco de datos de proteínas).

Cada proteína almacenada puede tener longitudes variables de aminoácidos. La proteína más corta tiene una longitud de 6 aminoácidos mientras que la más larga contiene 36.805 aminoácidos. En total la base de datos contiene 4.118.133.053 aminoácidos, lo que supone una media de 341 aminoácidos por proteína. En la siguiente gráfica (figura 3.3) se puede observar como la mayor parte de las proteínas de la *database* NR tienen una longitud entre 100 y 500 aminoácidos.

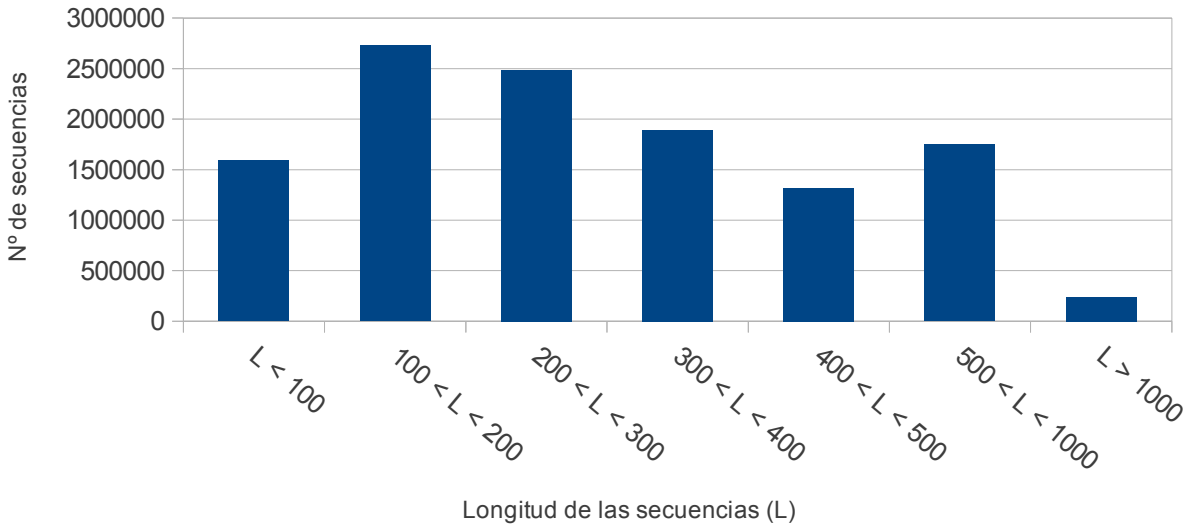


***Figura 3.3:*** *Tamaño de las secuencias en la base de datos NR*

### 3.1.3 Número de threads

También estudiaremos el comportamiento del programa cuando se usa el argumento opcional *-num_threads*, con el cual podemos escoger el número de threads que se van a ejecutar en *blastp.* De esta manera también podemos hacer un estudio comparativo para ver y estudiar como funciona y qué mejoras presenta la versión multi-core creada por el NCBI.

La aplicación *blastp* puede lanzar varios hilos de trabajo paralelos durante la fase nº 2 comentada en el capítulo anterior. Todos ellos realizan el mismo trabajo sólo que con distintos datos. La versión multi-core divide la base de datos haciendo que cada thread acceda sólo a una parte de ella. De esta manera se compara la secuencia *query* con las secuencias conocidas de forma paralela si se dispone de un procesador multi-core.

## 3.2 Entorno experimental y metodología

### 3.2.1 Descripción del H/W y del S.O

Para el análisis de rendimiento que se detalla a continuación se ha hecho uso de un portátil con un procesador Intel Core2 compuesto por 2 núcleos con una memoria RAM de 1 GB. Hay que mencionar que para tomar las medidas de este capítulo no se ha hecho uso de la batería, sino que en todo momento se ha usado la alimentación de red para evitar así la reducción de frecuencia de CPU que ocurre al trabajar en modo batería para ahorrar en consumo.

Por otro lado, para analizar con más profundidad la versión multi-core de *blastp* se ha usado una computadora con procesador Intel Core2 Quad con 4 núcleos y 4 GB de memoria RAM.

Las especificaciones de las computadoras se muestran a continuación:

***Computadora 1***

| **Hardware** | |
|---|---|
| CPU | Intel Core2 T5200 1.60Ghz (2 cores) |
| Cache L1 | 64 KB |
| Cache L2 | 2 MB |
| RAM | 1 GB |

| **Sistema Operativo** |
|---|
| Ubuntu 10.10 (Maverick) (32 bits) |
| Núcleo Linux 2.6.35-22-generic |

| **Software** |
|---|
| BLAST+ 2.2.24 (lanzado en Agosto 2010) |
| PAPI 4.4.1 (lanzado en Octubre 2010) |

***Computadora 2***

| **Hardware** | |
|---|---|
| CPU | Intel Core2Quad Q9400 2.66Ghz (4 cores) |
| Cache L1 | 64 KB |
| Cache L2 | 6 MB (2 x 3 MB, 1 caché L2 para 2 cores) |
| RAM | 4 GB |

| **Sistema Operativo** |
|---|
| Debian GNU 5.0.4 (64 bits) |
| Núcleo Linux 2.6.32-3-generic |

| **Software** |
|---|
| BLAST+ 2.2.24 (lanzado en Agosto 2010) |
| PAPI 3.7.1 (lanzado en Noviembre 2009) |

***Figura 3.4:*** *Características de las computadoras usadas para el análisis de rendimiento.*

### 3.2.2 Métodos para tomar las medidas

Las medidas tomadas sobre la aplicación han sido sobre la ejecución del programa en su totalidad, es decir, sin excluir operaciones de entrada/salida, inicialización de datos, etc. Para esto se ha hecho uso del comando *time,* de contadores hardware (PAPI) y de un profiler (*gprof)*.

- Time: se trata de un comando de los sistemas operativos Unix mediante el cual se obtienen estadísticas sobre el tiempo de ejecución de un proceso. Cuando dicho proceso termina su ejecución, se muestra un informe con el tiempo total de respuesta, el tiempo de cómputo y el tiempo de llamadas al sistema.

- PAPI (Performance Application Programming Interface): es una API para acceder a los contadores hardware de rendimiento, disponibles en la mayoría de los procesadores actuales. Estos contadores son un conjunto de registros que cuentan la ocurrencia de determinados eventos en el procesador. Al incorporar PAPI a una sección de código, es posible pasar (como parámetro) a la aplicación los nombres de los contadores hardware de los que se quiere obtener información. Para realizar las medidas de este capítulo se ha usado la versión más reciente (4.1.1) y se han usado los siguientes contadores:

  - PAPI_TOT_CYC Número total de ciclos de la aplicación. Se usará para calcular el tiempo de cómputo de las ejecuciones.
  - PAPI_TOT_INS Número total de instrucciones ejecutadas. Se puede usar para calcular CPI y TPI.
  - PAPI_L2_TCA Número de accesos a la caché L2. Para estudiar el uso de la caché.
  - PAPI_L2_TCM Número de fallos caché L2. Para calcular el porcentaje de fallos.
  - PAPI_TLB_DM Número de fallos en TLB de datos.
  - PAPI_TLB_IM Número de fallos en TLB de instrucciones.
  - PAPI_BR_CN Número de instrucciones de salto ejecutadas.
  - PAPI_BR_MSP Número de saltos mal predichos.

- Profiler: se ha hecho uso de la herramienta *gprof*, que permite realizar un análisis de rendimiento midiendo el comportamiento del programa mientras está en ejecución, particularmente la frecuencia y duración de las llamadas a funciones.

## 3.3 Experimentos

A continuación se muestran los experimentos realizados para observar el rendimiento de la aplicación. Se ha realizado primero un estudio sobre el tiempo de lectura de la *database,* ya que se espera que sea un factor importante en el rendimiento. Después se han realizado distintos experimentos diferenciados en cuatro apartados según: la longitud de las *queries,* la información proteica de las *queries,* el número de *queries* y el número de threads.

Las métricas usadas son varias:

- De los datos extraídos mediante PAPI se van a mostrar: ciclos de reloj, número de instrucciones, número de saltos, accesos a L2, fallos de caché, fallos de TLB, etc.

- Para mostrar tiempos de respuesta de *blastp* se usan segundos. En algunas ocasiones se mostraran los segundos de ejecución total de la aplicación (Tiempo) y en otras la suma del tiempo de cómputo y de llamadas a sistema (Tc+Ts). Esta diferenciación se realiza para observar si la lectura de la *database* provoca esperas de E/S que afecten en el rendimiento. Además se va a mostrar la media de tiempo por aminoácido de la secuencia query, medida en micro-segundos.

- De los datos obtenidos mediante el profiler *gprof* se van a mostrar la duración en segundos de las funciones con mayor consumo.

### 3.3.1 Estimación empírica del tiempo de lectura de la *database*

Se ha procedido a estudiar el tiempo de la lectura de la base de datos ya que se espera que sea un factor importante en el rendimiento. En los computadores utilizados no se tiene memoria suficiente para almacenar el tamaño de la *database*, así que durante la ejecución *blastp* se queda sin datos en memoria y hay que esperar que se traigan nuevos datos del disco. De esta forma, cada vez que se necesita traer datos de la *database* de disco a memoria, la CPU debe esperar por esos datos y así poder seguir con la ejecución. Este hecho puede provocar importantes retrasos en la ejecución ya que puede ocurrir que se tarde más en traer los datos a memoria que en realizar las operaciones con estos datos.

A continuación se muestra un ejemplo gráfico de este problema (figura 3.5).

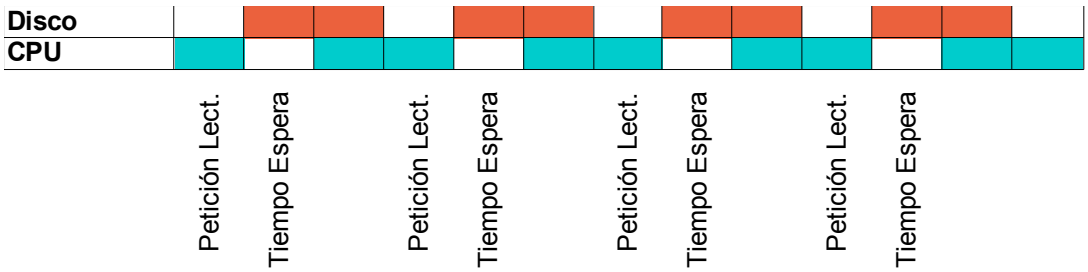


***Figura 3.5:*** *Ejemplo de tiempo de espera de CPU*

En este ejemplo se puede observar como la CPU realiza sus operaciones y hace una petición de lectura a memoria. Como en memoria no hay los datos necesarios se debe realizar una lectura a disco. Mientras se traen de disco a memoria los datos que necesita la CPU para continuar, esta se encuentra en espera. Una vez la CPU retoma su trabajo el disco continua trayendo más datos a memoria hasta que la CPU de nuevo se encuentra que debe esperar a que el disco traiga nuevos datos para seguir trabajando.

Para estudiar el tiempo de lectura de la *database* se ha realizado un experimento en el que se ha medido el tiempo de lectura de un fichero. Para ello se ha procedido a medir el tiempo de ejecución de un pequeño programa escrito en C que abre un fichero y lee su contenido. Aprovechando que la *database* está compuesto por diferentes ficheros, se ha medido el tiempo de lectura de uno de estos ficheros. El fichero seleccionado ha sido “nr.00.phr”, se trata de un archivo de 1.302 MB.

Después de 5 pruebas en la computadora de 2 núcleos, el tiempo de lectura ha sido de 58 ± 2.74 segundos. Por lo que el sistema lee a un ritmo aproximado de 22 MB/s. De esta forma podemos

calcular que la *database* que ocupa 9.5GB necesita sobre unos 442 segundos (7 minutos y 22 segundos) para ser leída completamente. En la computadora de 4 núcleos, al tener mejores prestaciones, necesita sobre unos 84 segundos (1 minuto y 24 segundos) para realizar la lectura de la *database*, con un ritmo aproximado de 118 MB/s. Este dato significa que cualquier ejecución de *blastp* tendrá como tiempo de respuesta mínimo estos tiempos calculados antes.

En los siguientes apartados veremos cómo hay ejecuciones menores a este tiempo, por lo que deducimos que *blastp* no usa toda la información almacenada en la *database*. La *database* contiene información y atributos para cada secuencia que *blastp* no necesita para su ejecución como pueden ser: número y fecha de patente, descripción de la secuencia, tipo de proteína, tipo de organismo a la que pertenece, nombre del organismo, función, etc.

### 3.3.2 Efecto de la longitud en el rendimiento

En este apartado el objetivo es identificar los cambios en el rendimiento de *blastp* cuando variamos la longitud de las *queries*. Cuanto mayor es el tamaño de la *query*, mayor es la cantidad de instrucciones a ejecutar. Mediante este experimento podremos observar la forma en que se incrementa la cantidad de cómputo de la aplicación al aumentar la longitud de la *query*.

A continuación se muestra una gráfica sobre el tiempo de ejecución de *blastp* con las distintas *queries* antes mencionadas en la figura 3.1. En ella se puede ver el tiempo total de ejecución (Tiempo) y la suma del tiempo de cómputo y de llamadas a sistema (Tc+Ts) (figura 3.6):

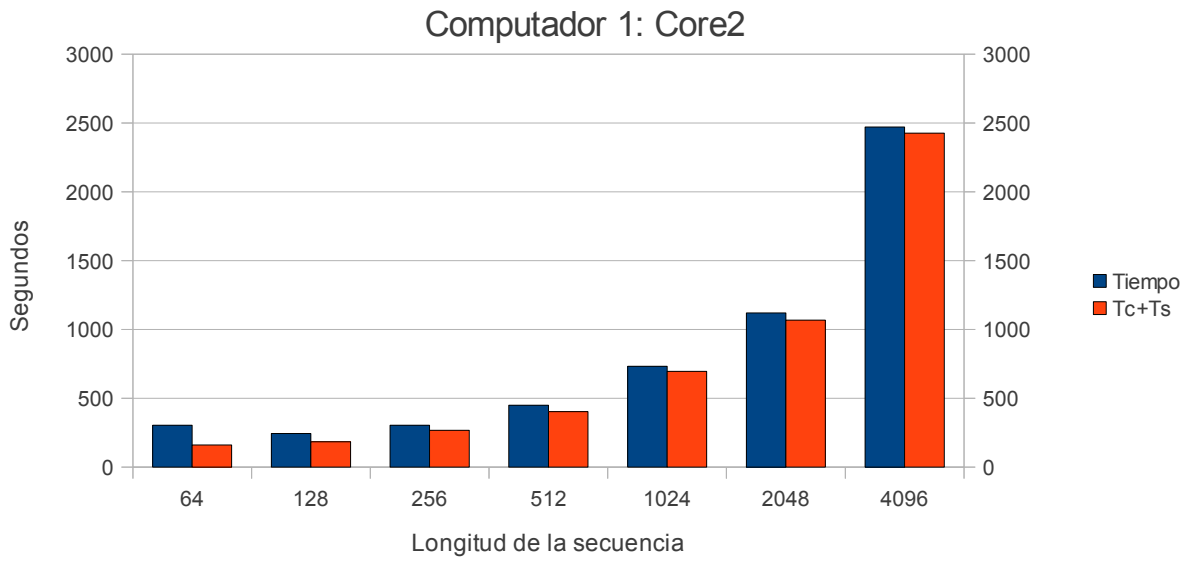


***Figura 3.6:*** *Tiempos de ejecución con queries de distintos tamaños*

Al incrementar la longitud de las *queries* nos encontramos que el tiempo de ejecución, en general, también se incrementa debido al incremento de trabajo a realizar. Se debe mencionar que los incrementos de tiempo no son del todo lineales, debido al factor antes mencionado sobre la información proteínica de cada *query*, hay que recordar que las secuencias usadas en este apartado son todas distintas entre ellas y puede ser que algunas *queries* requieran de más cómputo que otras sin tener en cuenta su tamaño.

En la gráfica se puede observar la diferencia entre el tiempo total de ejecución y el tiempo de cómputo (ya sea de CPU o de llamadas al sistema). No se trata de una diferencia que aumente al incrementar el tamaño de la *query*, se trata de una diferencia que parece ser constante. Es decir, no tiene relación ninguna con la longitud de la secuencia de entrada.

Se trata del tiempo de espera al disco mencionado en el apartado anterior. Hay momentos en el que la CPU se queda sin datos para procesar y tiene que esperar a que el disco traiga más datos a memoria. Estas esperas a disco provocan un retraso en el tiempo de ejecución que observando la gráfica podemos apreciar que no aumenta al incrementar la longitud de la *query.*

El hecho de que este tiempo de espera no aumente se debe a que cuanto mayor es el tamaño de la *query* más cantidad de cómputo tiene que realizar la CPU dando una cierta ventaja al disco que hace disminuir esta espera.

De esta forma podemos analizar que cuanto mayor es el tamaño de la *query* mayor es la reducción del porcentaje de tiempo de espera. Así que para *queries* de pequeño tamaño tendremos mucho tiempo de espera a disco, mientras que para *queries* de gran tamaño una mayor parte del tiempo de acceso a disco queda oculto, ya que la CPU tarda más en procesar los datos.

También podemos ver este fenómeno si observamos el tiempo medio que se tarda por aminoácido de la *query* de entrada (figura 3.7):

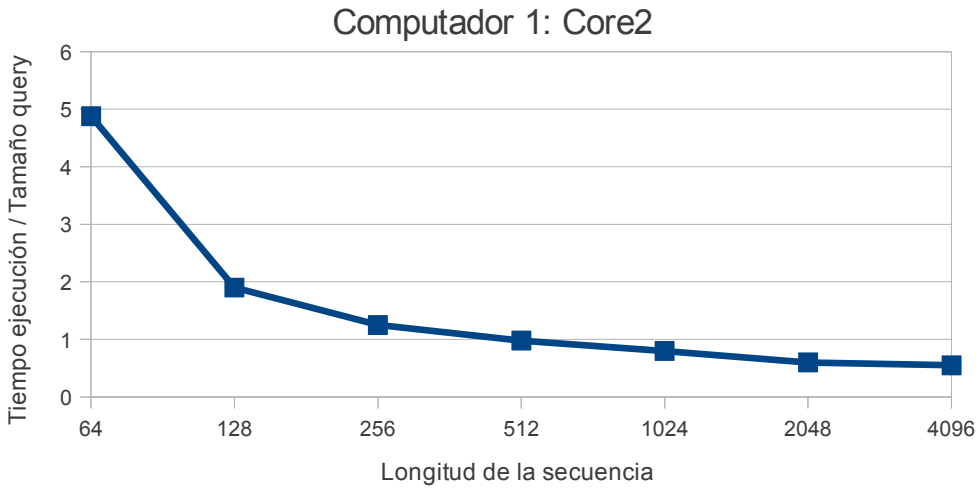


***Fig. 3.7:*** *Segundos por aminoácido para queries de distinto tamaño*

Se puede apreciar como en las *queries* de menor tamaño el tiempo medio por aminoácido es mucho mayor. Pero a partir de *queries* de unos 2048 aminoácidos se observa como el tiempo por aminoácido apenas se reduce, ya que el tiempo de espera a disco disminuye, quedando solapado por el aumento de cómputo.

A continuación se muestra el CPI (ciclos por instrucción) de la aplicación según variamos la longitud de las *queries* (figura 3.8). El CPI mostrado incluye la parte de cómputo de la ejecución excluyendo los tiempos de espera por el disco.

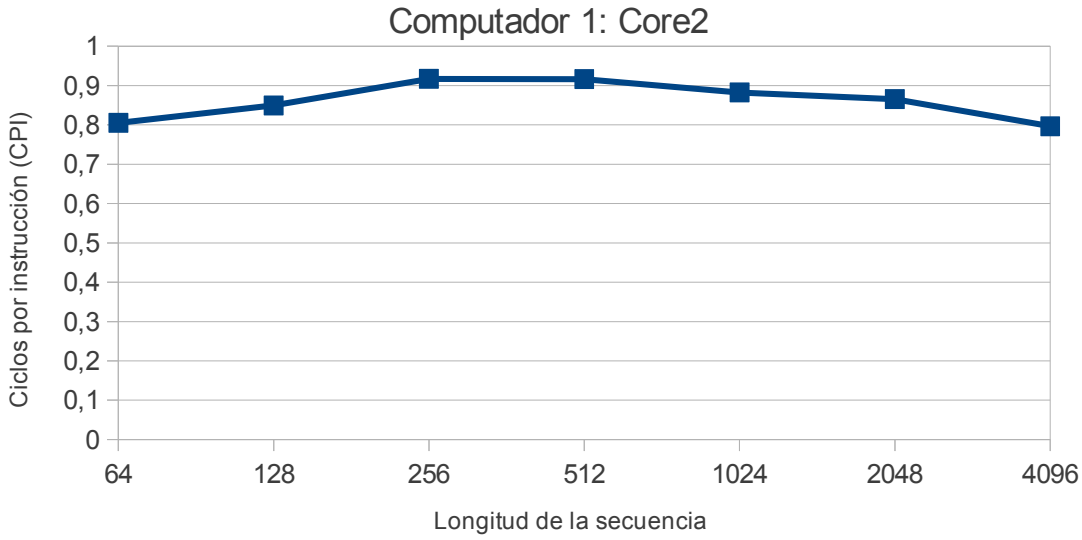


***Fig. 3.8:*** *Ciclos por instrucción según longitud de la query*

Se observa que el CPI ronda valores entre 0.8 y 0.9 ciclos por instrucción. Este valor no es muy bueno ya que el mejor CPI teórico posible en los computadores usados en el análisis es de 0.333 ciclos por instrucción. Este hecho da a entender que existe algún factor que no permite ejecutar varias instrucciones a la vez en cada ciclo. Este valor de CPI superior al valor ideal puede ser provocado por dependencias de datos (especialmente accesos a memoria que fallan en caché), dependencias de control (fallos de predicción de saltos) o saturación de ciertos recursos (unidades de cómputo o de acceso a memoria).

Si se observa el porcentaje de accesos a L2 (o lo que es lo mismo, fallos de caché L1) por instrucción (figura 3.9), se puede ver una posible correlación de la curva con la gráfica anterior.

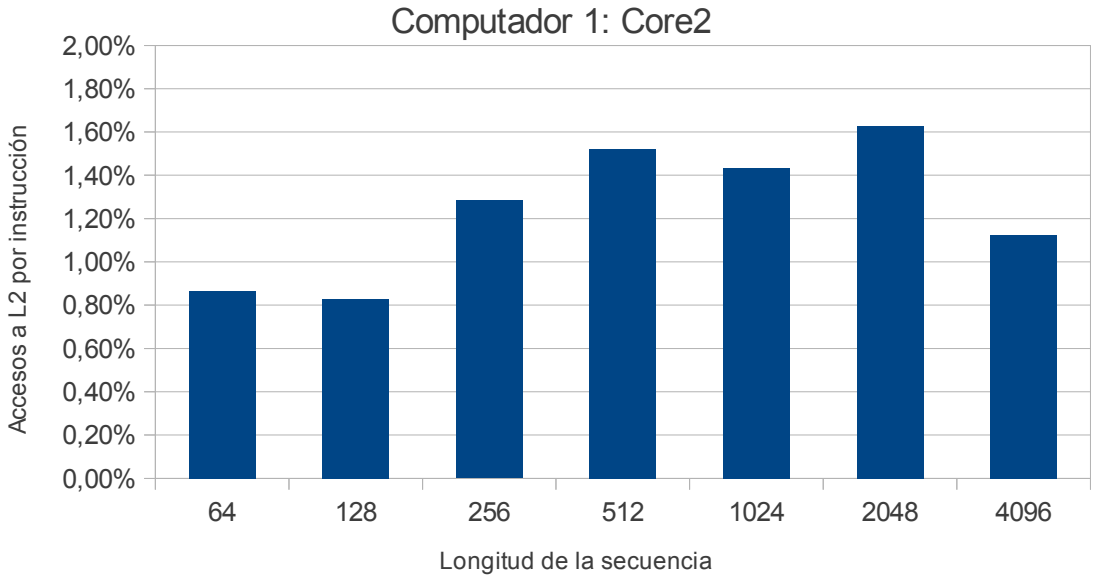


***Fig. 3.9:*** *Porcentaje de accesos a caché L2 por instrucción ejecutada*

Además, podemos observar el porcentaje de saltos mal predichos por instrucción (figura 3.10):

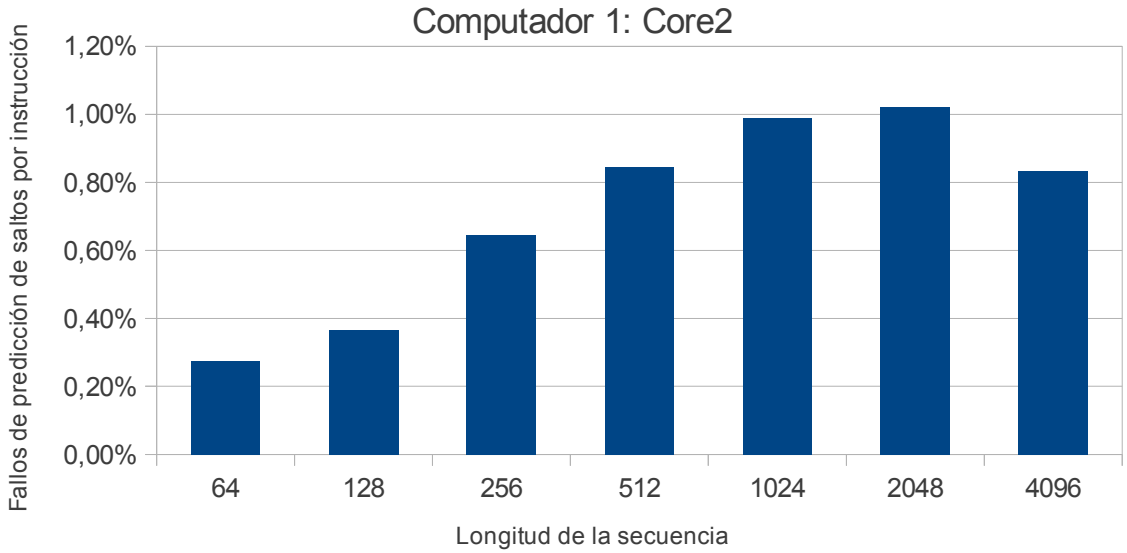


***Fig. 3.10:*** *Porcentaje de saltos mal predichos por instrucción*

También se puede apreciar la posible correlación con el CPI. La penalización que asocia una predicción incorrecta de salto afecta al rendimiento y a la vez a su CPI.

Si analizamos el número de fallos en la caché L2, o lo que es lo mismo, el número de accesos a memoria, se observa un ligero aumento de fallos al incrementar el tamaño de la *query* (figura 3.11). La mayoría de estos accesos a memoria corresponden a fallos a la hora de leer las secuencias de la *database.*

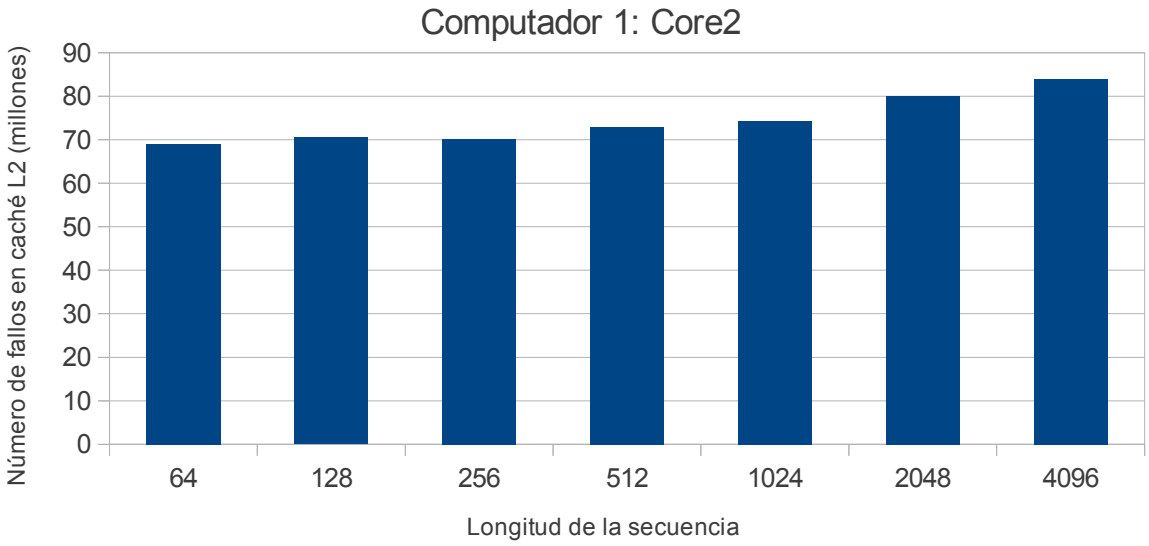


***Fig. 3.11:*** *Número de fallos en caché L2 usando queries de distinto tamaño (en millones)*

Este hecho provoca el porcentaje de accesos a memoria por instrucción decrezca como se puede observar en la siguiente gráfica (figura 3.12):

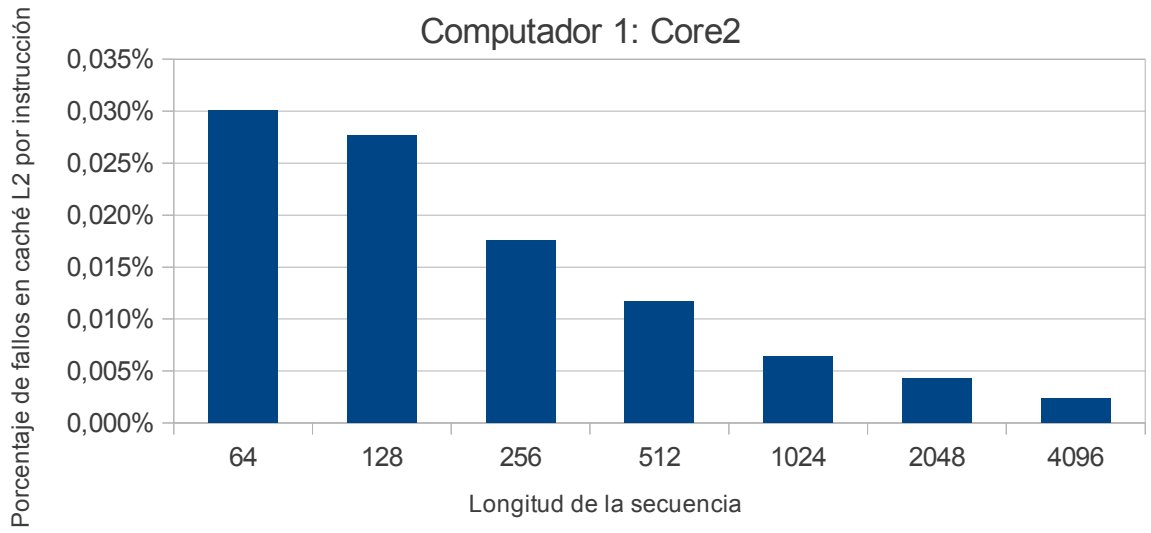


***Fig. 3.12:*** *Porcentaje de accesos a memoria por instrucción ejecutada*

Finalmente, se estudia el consumo de recursos de cada fase de la que consta la aplicación. Se obtiene que la fase con más peso o consumo en tiempo y recursos es la fase nº 2 (fase de *scanning*: compara la *query* con las secuencias de la base de datos y realiza las extensiones). A continuación podemos ver el porcentaje de tiempo que consumen las 3 fases de las que consta *blastp* en función de la longitud de secuencia (figura 3.13):

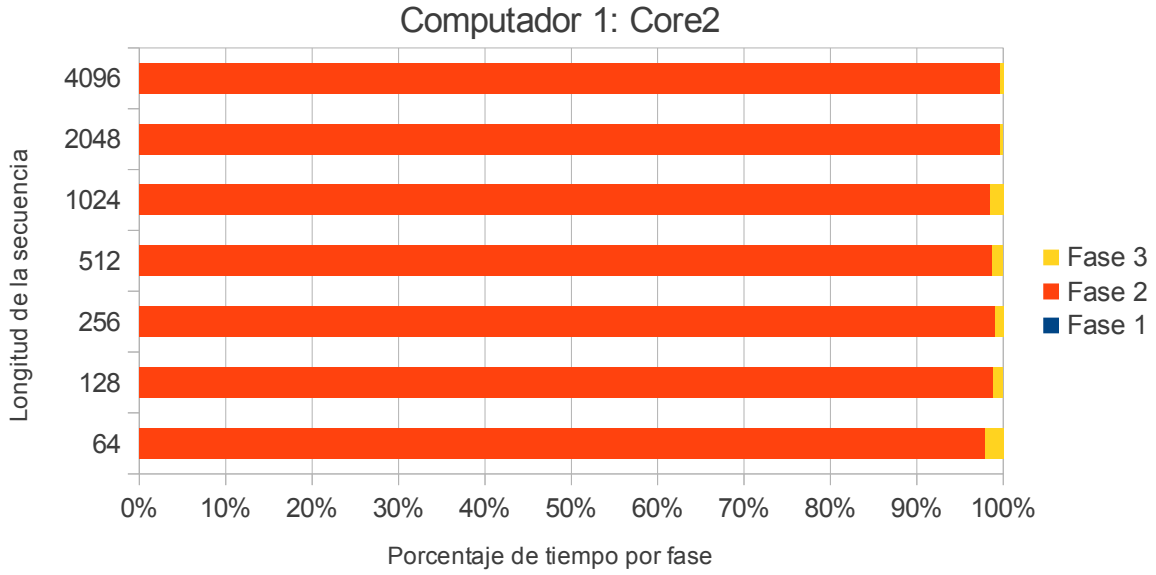


***Fig. 3.13:*** *Porcentaje de tiempo para cada fase de blastp*

Observando el gráfico se puede ver como la fase nº 1 (s*etup*) es inapreciable y se trata de una fase que podemos despeciar en el estudio. La fase nº 3 (*trace-back*) aunque no es despreciable, consume poco tiempo y cuánto mayor es el tamaño de la *query* menor es su peso en el rendimiento. La fase nº 2 (*scanning*) es la fase que más cómputo necesita y la fase a atacar en futuras optimizaciones. Se trata de la fase más dominante en la ejecución y su peso e importancia aumenta a la vez que crece la longitud de la secuencia *query*.

### 3.3.3 Efecto de la información proteínica en el rendimiento

Como ya se ha comentado en el apartado 3.1.1, el contenido de una secuencia repercute tanto en el tiempo de cómputo como en el espacio de memoria usado. Se espera que el contenido de cada *query* pueda hacer variar la cantidad de trabajo. Cuanta más diversificación en las *words* más *words* hay que almacenar en la LUT provocando un aumento de su tamaño, y además, si da el caso que hay *words* muy comunes en la base de datos se tendrán que almacenar más *hits* en memoria. Ambos casos pueden provocar una tasa mayor de fallos de accesos tanto en caché como en memoria (accesos a la tabla *hash* de la LUT). En los siguientes experimentos se analiza la variabilidad de los tiempos de ejecución, número de ciclos, instrucciones, etc. con distintas *queries* del mismo tamaño.

Vamos a tomar las medidas del conjunto de 16 *queries* mostradas en la figura 3.2. Como todas tienen la misma longitud, las diferencias encontradas se deberán únicamente a su contenido. En la siguiente figura se muestran los tiempos de ejecución de las *queries* ordenado de menor a mayor incluyendo los tiempos de E/S (figura 3.14):

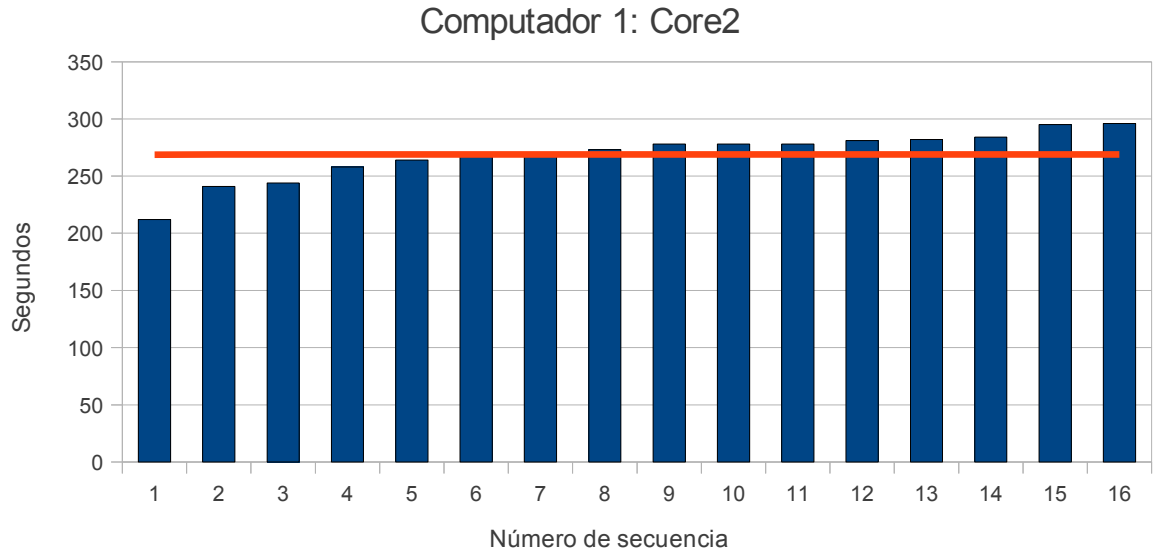


***Figura 3.14:*** *Tiempos (con E/S) de distintas queries del mismo tamaño ordenadas por tiempo*

Se puede apreciar como el contenido proteínico afecta al tiempo de ejecución. Los tiempos han sido de 267 ± 20.3 segundos, desviación suficiente para comprobar este efecto. También se puede ver como hay *queries* con un tiempo cercano a 200 segundos y otras cercanas a los 300 segundos.

Si observamos el número de ciclos y de instrucciones se puede apreciar mejor la variabilidad que produce la información proteínica. En la siguiente figura se muestran ciclos e instrucciones de las *queries* ordenadas por tiempo de ejecución (figura 3.15):

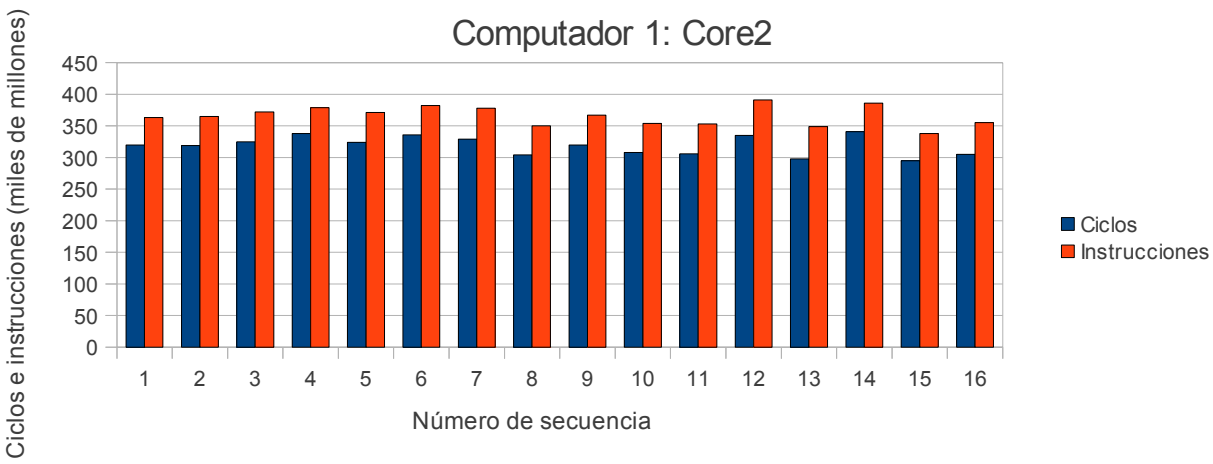


***Figura 3.15:*** *Número de ciclos e instrucciones de las queries ordenadas por tiempo (en miles de millones)*

Aunque en la gráfica anterior (figura 3.15 pueda parecer que hay mucha diferencia entre las *query* en cuanto a la proporción de ciclos e instrucciones, podemos ver como el CPI es muy similar. A continuación se muestra el CPI (figura 3.16):

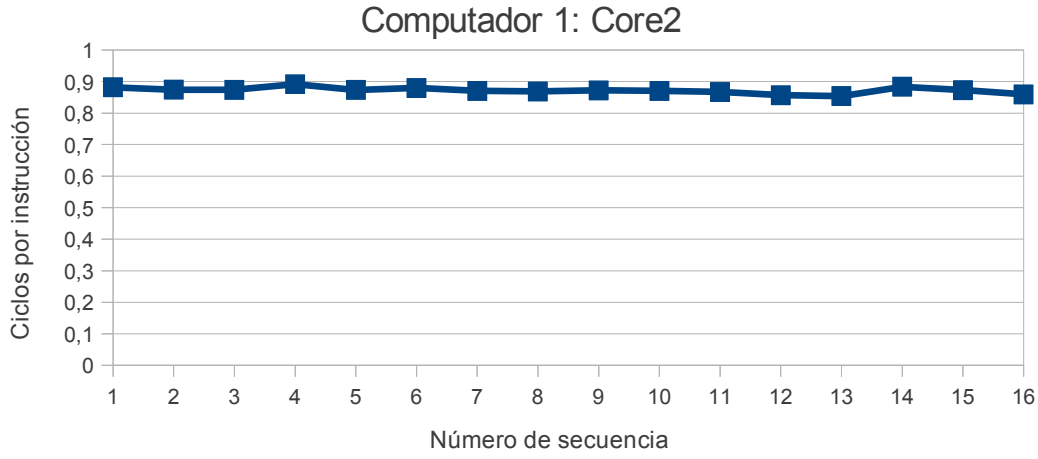


***Figura 3.16:*** *Ciclos y tiempo por instrucción*

Mientras que el CPI sin incluir los ciclos de E/S parece ser constante, el CPI incluyendo la E/S demuestra que la información proteínica que contiene una secuencia es un factor a tener en cuenta en el rendimiento. Si las subsecuencias o *words* de una *query* son muy o poco repetitivas a lo largo de la secuencia afecta en el tamaño de la LUT, provocando una tasa mayor de fallos de acceso. Además, al comparar con la *database* se almacenan distintas cantidades de *hits* dependiendo del contenido de la *query*, hecho que afecta también al rendimiento.

### 3.3.4 Efecto del número de queries en el rendimiento

Como ya se ha comentado en este capítulo, al ejecutar *blastp* podemos pasar como parámetro de entrada una sola *query* o varias de ellas. En este apartado se estudiará como repercute en el rendimiento el número de *queries*. Se analizará si es mejor hacer N búsquedas de una *query* o una búsqueda de N *queries*.

Se van a medir las 16 *queries* de la figura 3.2 por separado y, a continuación, se van a medir agrupándolas en grupos de 2, 4 y 8, hasta hacer una ejecución con las 16 *queries* a la vez. Se espera obtener un tiempo de ejecución menor al juntar distintas *queries* en una sola búsqueda ya que de esta manera solo se accede una vez a la *database* y no una vez para cada *query.*

En la figura 3.12 ya tenemos las medidas tomadas de las *queries* por separado, es decir, una ejecución de *blastp* por *query*. Así que ahora vamos a agrupar las *queries* en parejas y en orden de rapidez de ejecución, es decir, se va a lanzar *blastp* con las *queries* 1-2 (las más rápidas), 3-4,... y 15-16 (las más lentas). Los resultados se muestran a continuación (figura 3.17):

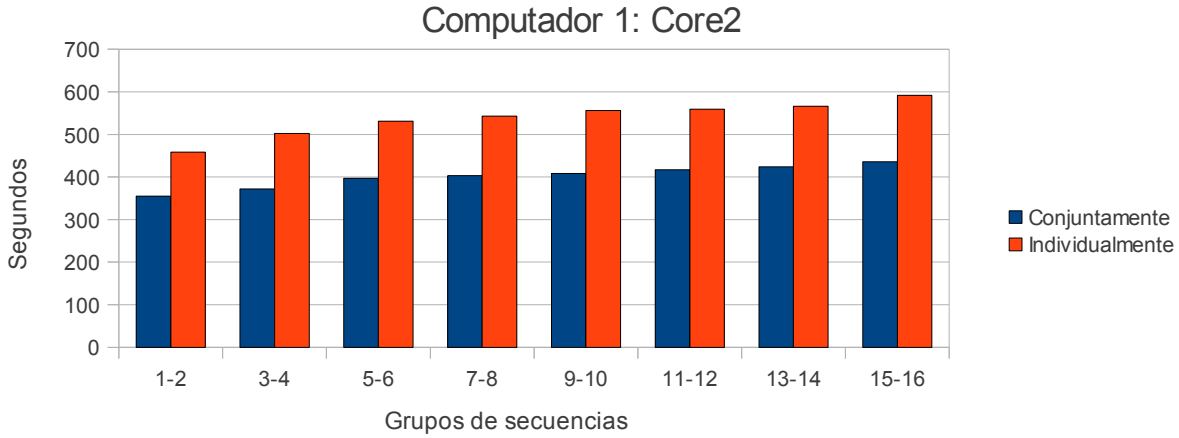


***Figura 3.17:*** *Tiempos de ejecución con E/S en grupos de 2 queries e individualmente*

Podemos observar cómo la aplicación tarda entre un 25-30% menos si trabajamos con dos *queries* a la vez y no por separado. Este hecho se produce porque *blastp,* sin tener en cuenta el número de *queries* que hay que tratar, solo hace una lectura de la *database.* Es decir, aprovecha cada dato que se lee de la *database* para usarlo con todas las *queries* que se le han pasado como entrada. De esta manera, se reducen las instrucciones a ejecutar en la aplicación (figura 3.18). Además, si tenemos más *queries* tenemos más cómputo, y así no hay tanta espera en la CPU por lecturas a disco.

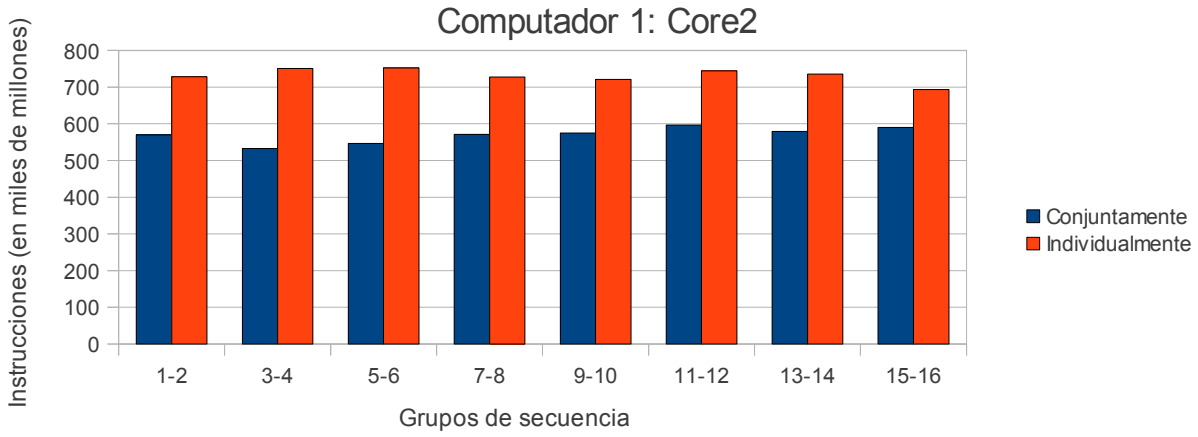


***Figura 3.18:*** *Número de instrucciones en grupos de 2 e individualmente (en miles de millones)*

A continuación podemos ver que cuantas más *queries* se tienen, menor es el tiempo de ejecución (figura 3.19):

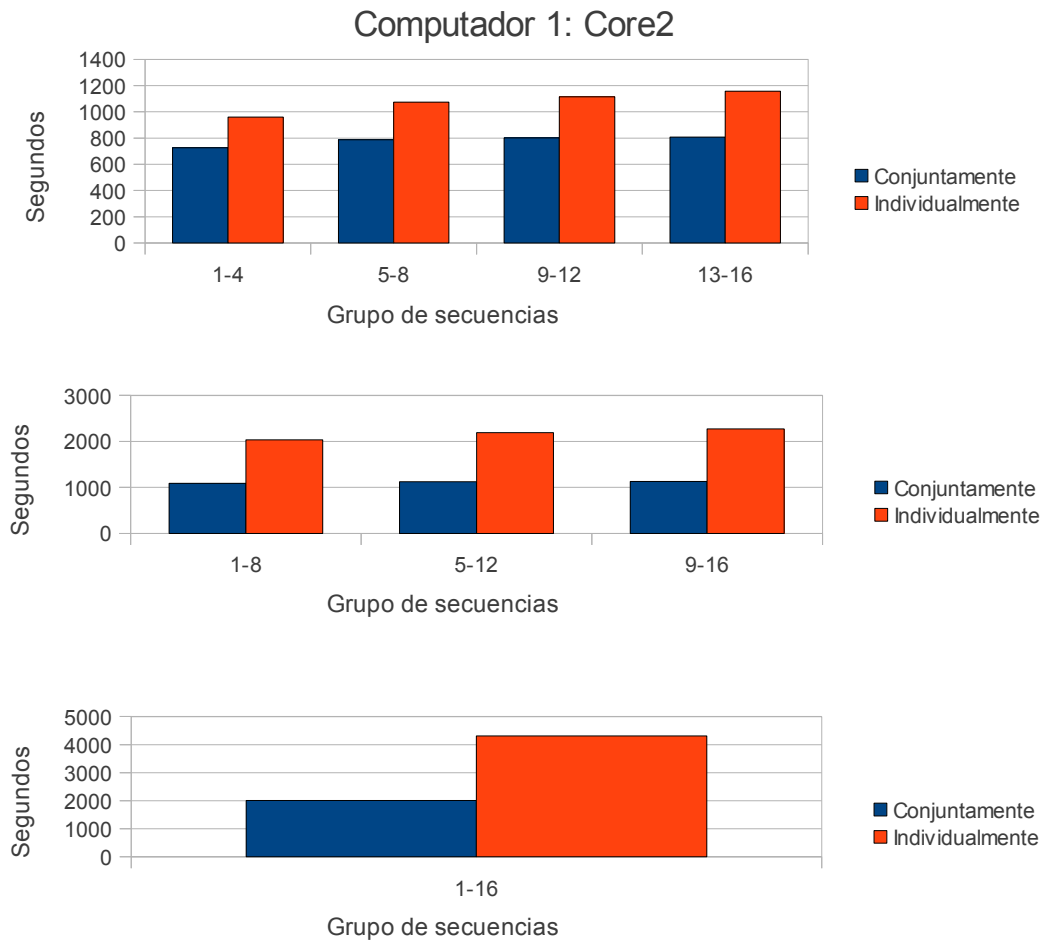


***Figura 3.19:*** *Tiempos de ejecución en grupos de 4, 8 y 16 queries e individualmente*

En la figura anterior se puede apreciar cómo al incrementar el número de *queries*, la aplicación tiene un tiempo de ejecución menor, llegando a reducir entre un 55-60% el tiempo si se ejecuta la aplicación con las 16 *queries* del conjunto de prueba a la vez.

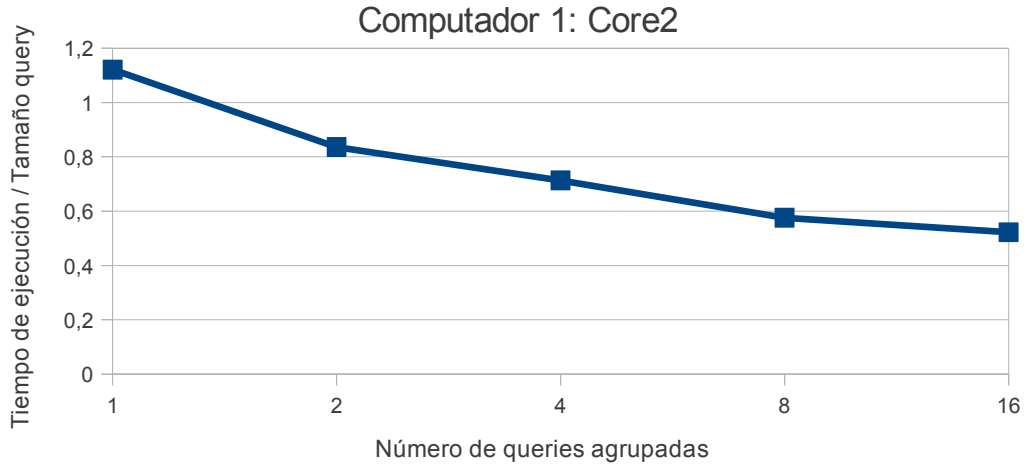


***Fig. 3.20:*** *Segundos por aminoácido para conjuntos de queries del mismo tamaño*

Si analizamos el tiempo medio que se tarda por aminoácido de las *queries* de entrada (figura 3.20), podemos apreciar como el tiempo va disminuyendo al agrupar más *queries*. Es decir, el hecho de agrupar las *queries* en una misma búsqueda resulta más efectivo que hacer varias búsquedas por la razón de que el contenido de la *database* se lee menos veces en total.

El CPI (figura 3.21) se comporta de igual manera que en el apartado 3.3.2. No se trata de un buen promedio ya que el CPI ideal es de 0,33 en las computadoras utilizadas. Puede ser provocado por dependencias de datos (especialmente accesos a memoria que fallan en caché), dependencias de control (fallos de predicción de saltos) o saturación de ciertos recursos (unidades de cómputo o de acceso a memoria).

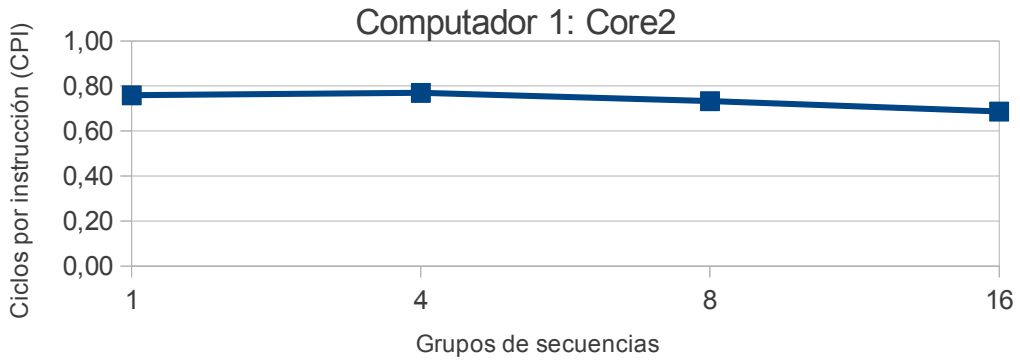


***Fig. 3.21:*** *CPI agrupando las queries*

### 3.3.5 Efecto del número de threads en el rendimiento

Al lanzar *blastp* podemos especificar el número de threads que se van a ejecutar. De esta manera se puede paralelizar parte del cómputo siempre que dispongamos de un procesador multi-core. El objetivo de este apartado es analizar el comportamiento de esta versión multi-core para encontrar posibles problemas de rendimiento.

De las tres fases de las que se compone la aplicación, cuando lanzamos *blastp* en su modo multi-core sólo se ejecuta de forma paralela la fase nº 2. Las fases nº 1 y 3 siempre se ejecutan de forma secuencial. La fase nº 2, comentada en el capítulo anterior, es la encargada de leer todas las secuencias conocidas en la *database* y hacer las comparaciones con las *queries*, almacenando *hits* que más tarde intentará expandir.

*Blastp* reparte el trabajo a los distintos threads lanzados, dándoles a cada thread distintos datos para realizar el mismo trabajo. Es decir, se divide la *database* en varios bloques o porciones que se reparten entre sus threads. De esta manera se puede paralelizar el cómputo significativamente. Cada thread compara la *query* o *queries* con su propio bloque de datos de la *database*. Cuando un thread termina de ejecutar un bloque pide el siguiente, mientras queden bloques disponibles.

A continuación se van a estudiar dos escenarios particulares que se espera que sean de importancia en el rendimiento de la versión multi-core creada por el NCBI: la forma en que se dividen los bloques y la asignación de los threads en los cores. Además de la computadora de 2 núcleos (Intel Core2), para analizar con más profundidad la versión multi-core de *blastp* también se ha hecho uso de una computadora con 4 núcleos (Intel Core2 Quad).

*División de los datos*

El hecho de tener la *database* almacenada en disco sin fragmentación es importante para la ejecución de *blastp*. Tener la base de datos desfragmentada reduce el tiempo de movimiento y rotación del cabezal del disco minimizando el tiempo de espera. Además se favorece la localidad temporal, ya que al pedir una secuencia a la *database*, cada bloque leído de disco contiene la secuencia que se ha pedido y varias secuencias más que se usarán en las siguientes iteraciones.

La forma en que la aplicación trata las secuencias de la *database* es mediante un bucle. El bucle tiene tantas iteraciones como secuencias conocidas existen en la base de datos. Cada iteración pide a la *database* la siguiente secuencia a tratar y la procesa, hasta que no quedan más secuencias a procesar.

Si medimos mediante la librería *time.h* el tiempo de ejecución de la función que lee las secuencias a procesar de la *database*, podemos observar importantes diferencias en el tiempo de lectura de las secuencias. La siguiente figura muestra en pseudocódigo el bucle mencionado (figura 3.22).

```
indice = siguienteSecuencia ( database ) ;
Mientras ( indice <> FIN )
        t1 = clock () ;
        secuencia = leerSecuencia ( indice , database ) ;
        t2 = clock () ;
        procesarSecuencia ( secuencia ) ;
        indice = siguienteSecuencia ( database ) ;
Fin Mientras
```

***Fig. 3.22:*** *Pseudocódigo de la lectura de secuencias de la database. En azul, la función encargada de leer la secuencia, y en rojo, los lugares en el código donde se han tomado las medidas de tiempo.*

Los tiempos tomados al leer una secuencia de la *database* en la versión secuencial se encuentran entre dos rangos diferentes: de 1 a 500 micro-segundos y de 1000 a 50.000 micro-segundos. Si estudiamos los tiempos superiores a 1000 micro-segundos, en la siguiente gráfica podemos observar como la versión multi-core resulta tener muchos más accesos con tiempos superiores a 1 mili-segundo que usando la versión secuencial (figura 3.23):

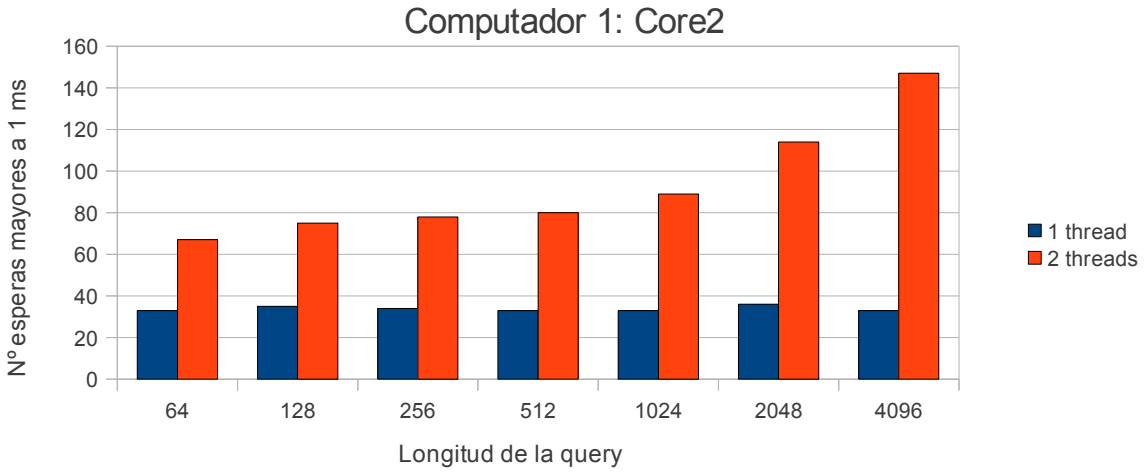


***Figura 3.23:*** *Número de esperas superiores a 1 ms (secuencial y multi-core)*

Podemos observar cómo en la versión secuencial el número de esperas superiores a 1 ms son constantes, sobre 30 ocasiones en cada ejecución. Mientras que en la versión multi-core (2 threads) el número de esperas es más elevado y además se incrementa al aumentar la longitud de la *query.*

Estos incrementos son debidos a la mala división que hace *blastp* de la *database* en la versión multi-core. De las más de 12 millones de secuencias que almacena la *database* usada en este análisis, la aplicación divide las secuencias en rangos demasiado grandes. Es decir, que los distintos threads piden datos que quizás se encuentran en diferentes sectores, hecho que provoca muchos cambios de cabezal en una computadora con un solo disco.

Si se muestran los tiempos máximos de acceso, se puede ver con más claridad como en la versión multi-core son mucho más grandes, llegando algunas veces a medio segundo (figura 3.24):

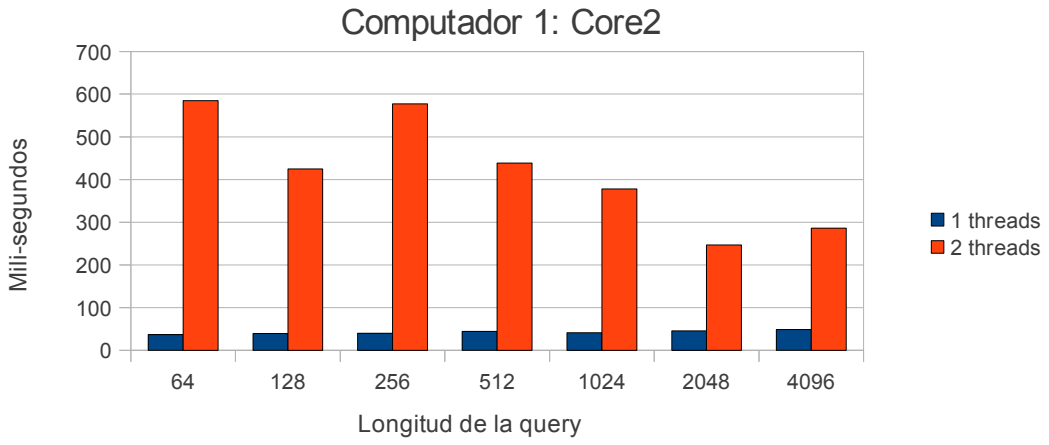


***Figura 3.24:*** *Tiempos máximos en mili-segundos de lectura de secuencias (secuencial y multi-core)*

Una posible solución a este problema puede ser repartir las secuencias en rangos más pequeños o quizás asignar en orden cada secuencia a un thread distinto. Otra posible solución a este problema es el uso de sistemas con varios discos, o con RAIDs, con los que se podría aprovechar mejor esta partición de la *database.*

Si se observa el porcentaje del tiempo total de ejecución que *blastp* dedica a la E/S (figura 3.25) se puede apreciar con más claridad este fenómeno:

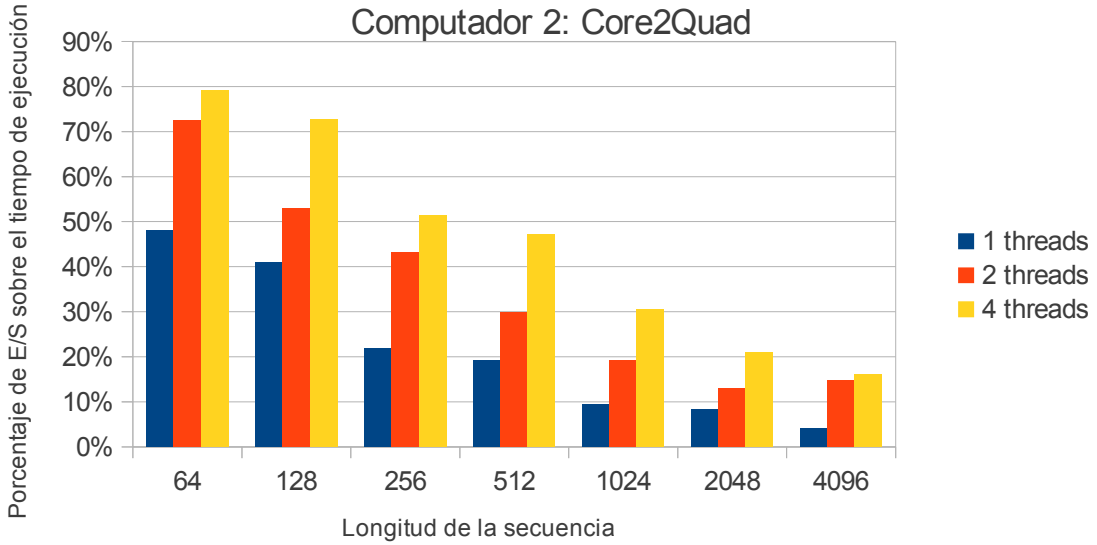


***Fig. 3.25:*** *Porcentaje de tiempo de ejecución que blastp dedica a E/S*

Existe una relación entre el número de threads lanzados y el tiempo que dedica la aplicación a la E/S. Así que hay que estudiar una manera de acceder a las secuencias en el modo multi-core de una forma más óptima.

Pero éste no es el único problema que presenta la división de datos de *blastp*. Además de dividir la *database* en bloques demasiado grandes que provocan mayores esperas a disco, estos bloques no son equitativos ni en número de secuencias ni en número de aminoácidos. Podemos observar que el número de ciclos consumidos por cada core del procesador no es equitativo en la fase nº 2 (figura 3.26):

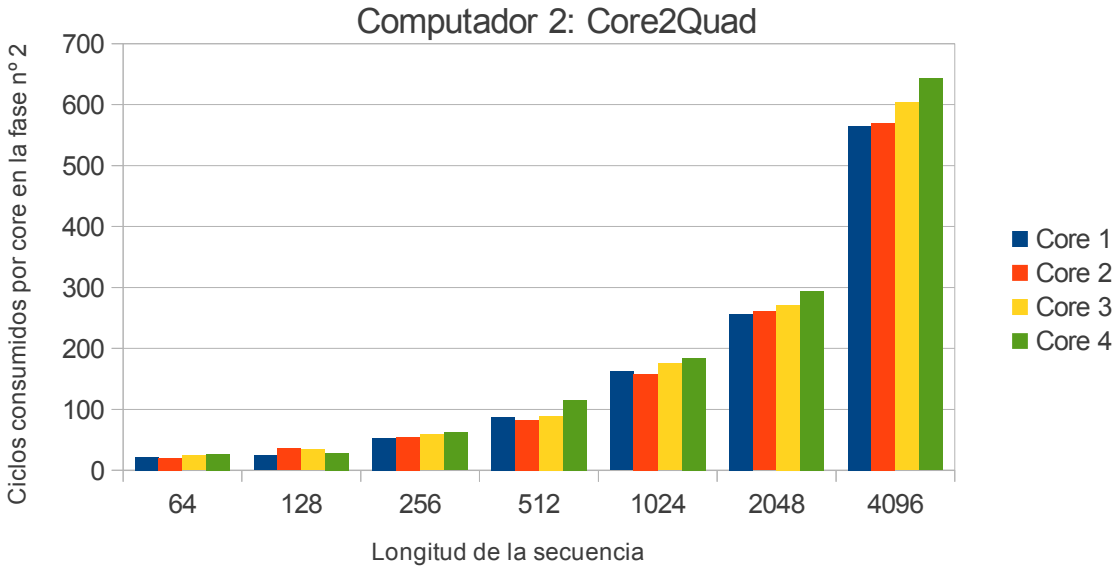


***Fig. 3.26:*** *Ciclos consumidos por core en la fase nº 2 (en miles de millones). No incluye tiempo de E/S.*

Este hecho provoca que los distintos cores tengan cargas de trabajo distintas, provocando que los threads con menos carga de trabajo queden a la espera del thread más lento y con más carga de trabajo. De esta manera, durante algunos instantes se tienen uno o varios cores desaprovechados. Así que hay que estudiar cómo repartir de forma equitativa el trabajo a realizar entre los distintos threads.

*Asignación de threads en los cores*

Cuando un proceso lanza varios threads, es interesante asignar cada thread a un core distinto para aprovechar el paralelismo en los sistemas multi-core. Generalmente, el sistema operativo es el encargado de asignar los threads en los cores. Así que, en una computadora con dos cores, si se lanzan dos threads y los dos se empiezan a ejecutar en el mismo core, el sistema operativo se encarga de hacer migrar uno de los dos al core que no se está usando o tiene menor carga de trabajo.

Pero a la hora de hacer el análisis en muchas ocasiones este hecho no sucedía así. Sólo en algunas ocasiones el consumo de CPU era más o menos equivalente en cada core, en el resto de ocasiones podríamos encontrarnos con un core con una carga de trabajo mucho mayor al otro. De esta manera, en algunos casos era más o igual de eficiente la versión secuencial que la multi-core, como se puede ver en la siguiente gráfica (figura 3.27):

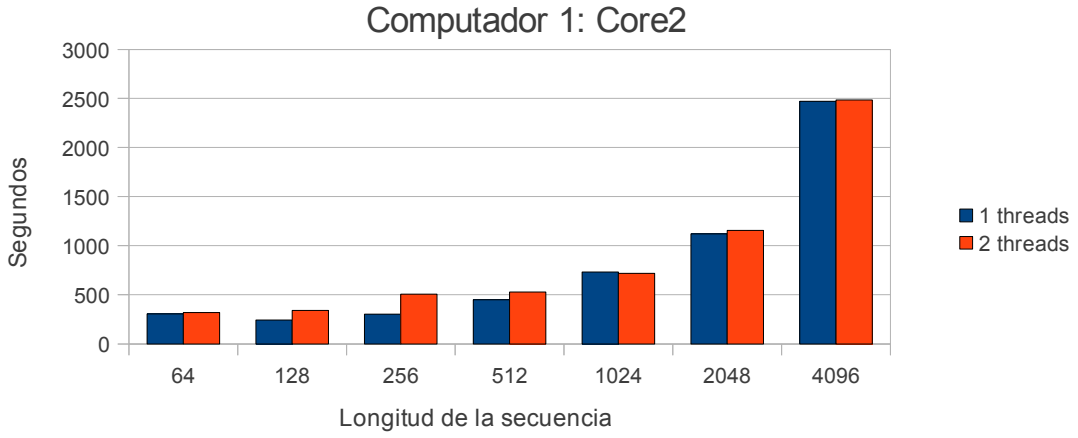


***Figura 3.27:*** *Comparación de tiempo con E/S en segundos (secuencial y con 2 threads)*

Se puede observar como el speedup es la mayoría de veces peor con dos threads. Si se fuerza a cada thread a ser ejecutado en un core distinto mediante la librería *sched.h*, el rendimiento mejora mucho, como se puede apreciar en la siguiente gráfica (figura 3.28):

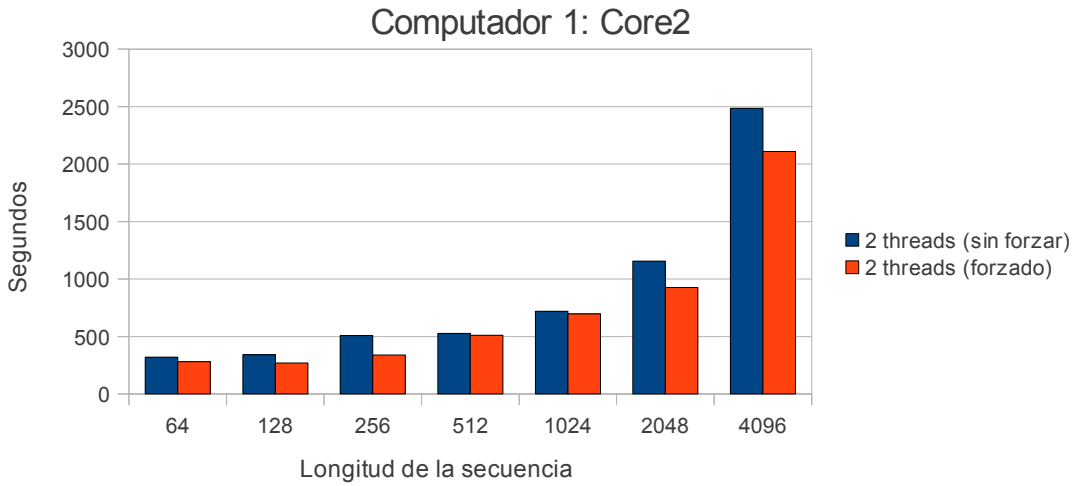


***Figura 3.28:*** *Comparación de tiempo con E/S en segundos de la versión multi-core forzando y sin forzar la ejecución de cada thread en un core*

Así que para obtener mayor rendimiento aprovechando al máximo el paralelismo de los sistema multi-core se puede forzar a cada thread a ser ejecutado en distinto core, siempre y cuando haya cores disponibles.

*Resultados de los experimentos multi-core*

Usando la librería *sched.h* para forzar a cada core a ejecutar un thread, se dispone a mostrar y comentar los resultados obtenidos al usar la versión multi-core de *blastp.*

En la siguiente gráfica podemos observar una comparativa de ciclos consumidos y instrucciones ejecutadas entre la versión secuencial y la multi-core, cuando tenemos una sola *query* de distintos tamaños (figuras 3.29 y 3.30):

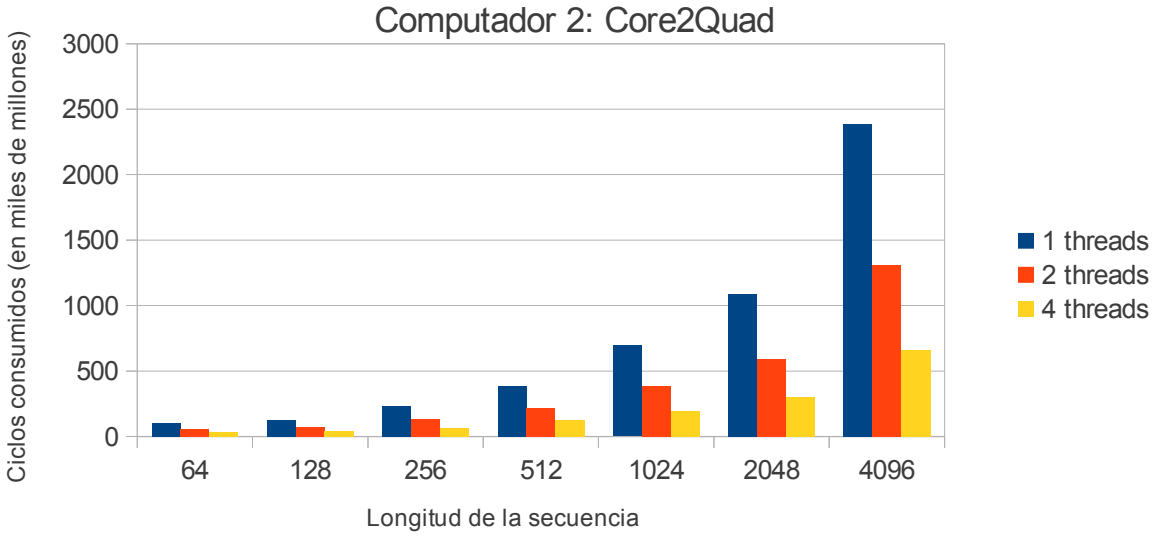


***Fig. 3.29:*** *Ciclos con una query de distintos tamaños usando varios threads (en miles de millones). No incluye tiempo de E/S.*

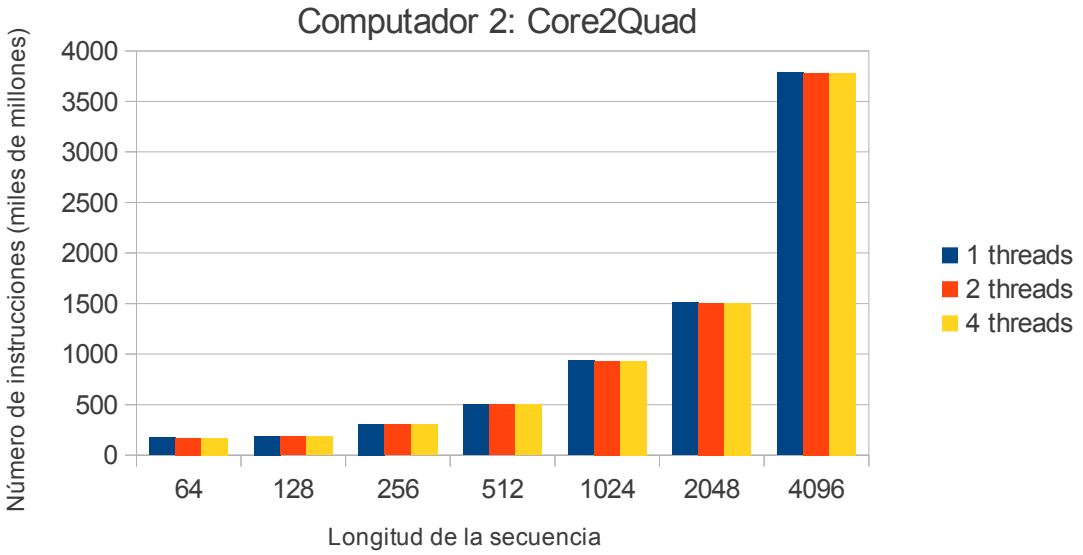


***Figura 3.30:*** *Total de instrucciones eejcutadas con una query usando varios threads (en miles de millones)*

Observando las gráficas podemos ver que tanto en la versión secuencial como en la multi-core se ejecuta el mismo número total de instrucciones, consumiendo menos ciclos de ejecución en la versión multi-core. El speedup calculado por ciclos es una mejora interesante (ver figuras 3.31 y 3.32), pero hay que recordar que, como se ha hablado en los apartados anteriores, la versión multi-core tiene un tiempo de espera a disco mayor. Esto provoca que el speedup de la versión multi-core sea mucho menor del esperado, llegando a ser en algunas ocasiones más rápida la versión secuencial.

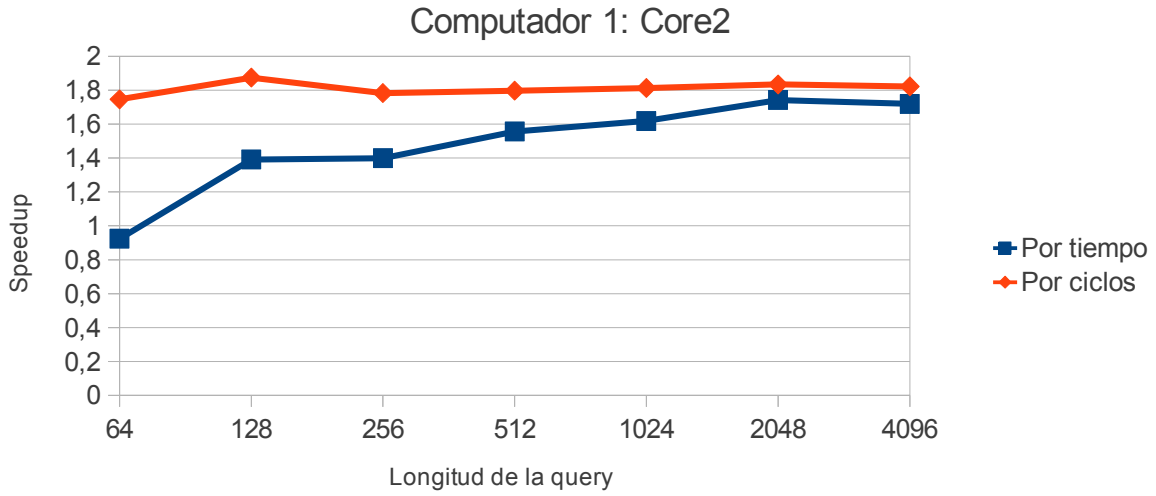


***Fig. 3.31:*** *Speedup entre las versiones secuencial y con 2 threads*

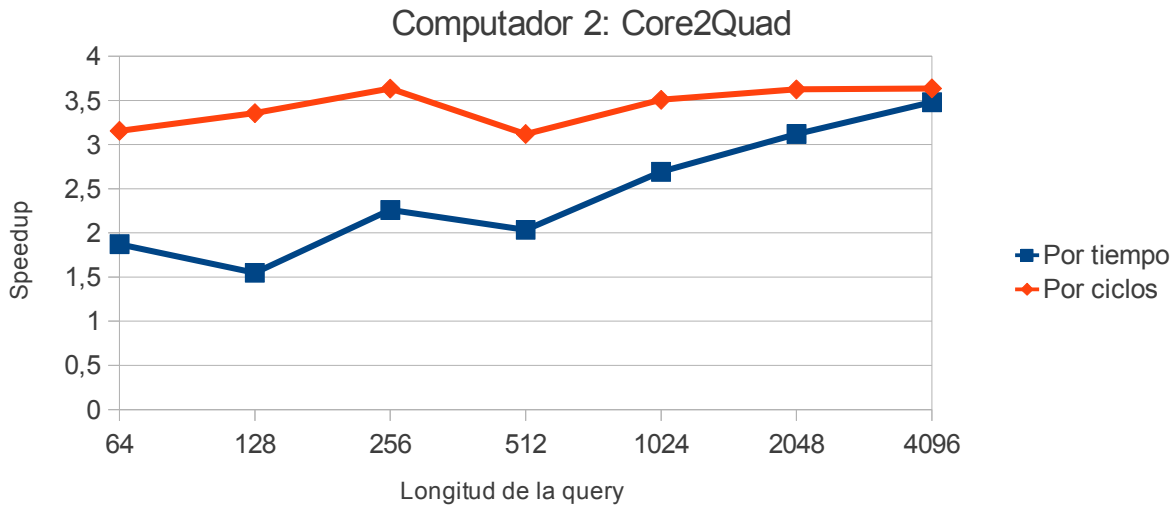


***Fig. 3.32:*** *Speedup entre las versiones secuencial y con 4 threads*

Podemos observar en las gráficas anteriores como las *queries* de menor longitud tienen un speedup peor. Es decir, en la versión multi-core, aquellas *queries* con menor cómputo son más penalizadas por los tiempos de espera a disco.

# Capítulo 4

# Optimizaciones

En este capítulo se muestran las distintas optimizaciones que han sido implementadas en *blastp* y los resultados obtenidos, que van a ser comparados con el rendimiento de la aplicación original. Aunque se trata de una compleja aplicación, gracias al análisis de rendimiento realizado anteriormente, podemos mejorar algunos aspectos que afectan de forma positiva en el rendimiento. Estas optimizaciones se pueden clasificar en dos tipos: para sistemas con un único procesador y para sistemas multi-core.

## 4.1 Optimizando la aplicación para un procesador

Al realizar el estudio de *blastp*, se puede apreciar a simple vista que se trata de una aplicación compleja y de grandes dimensiones. Cuando hay que realizar optimizaciones es difícil modificar una aplicación de estas magnitudes. Así que se deben estudiar aquellas partes del código más significativas, analizar las funciones ejecutadas más veces, las que demoran más en su tiempo de respuesta, etc. Se deben encontrar aquellas partes del programa que ralentizan el proceso, que representan un cuello de botella y se debe analizar si pueden ser optimizadas.

Si ejecutamos *blastp* con las 7 *queries* usadas en el capítulo anterior (figura 3.1) por separado, mediante el profiler *gprof* podemos obtener aquellas funciones del código de la aplicación que acumulan un tiempo total de ejecución mayor. Las dos funciones que más tiempo de ejecución consumen son *s_BlastSmallAaScanSubject* y s_*BlastAaWordFinder_TwoHit*, que llamaremos a partir de ahora *SCAN* y *FIND* respectivamente. *S*us porcentajes de tiempo de ejecución sobre la ejecución total del proceso son los siguientes (figura 4.1 y figura 4.2):

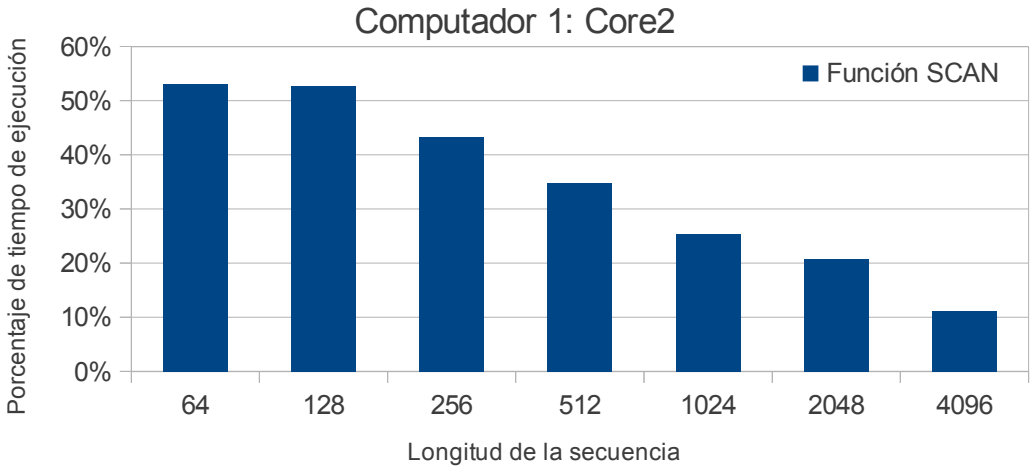


***Fig. 4.1:*** *Porcentaje de tiempo de ejecución de la función 's_BlastSmallAaScanSubject' sobre la ejecución de blastp.*

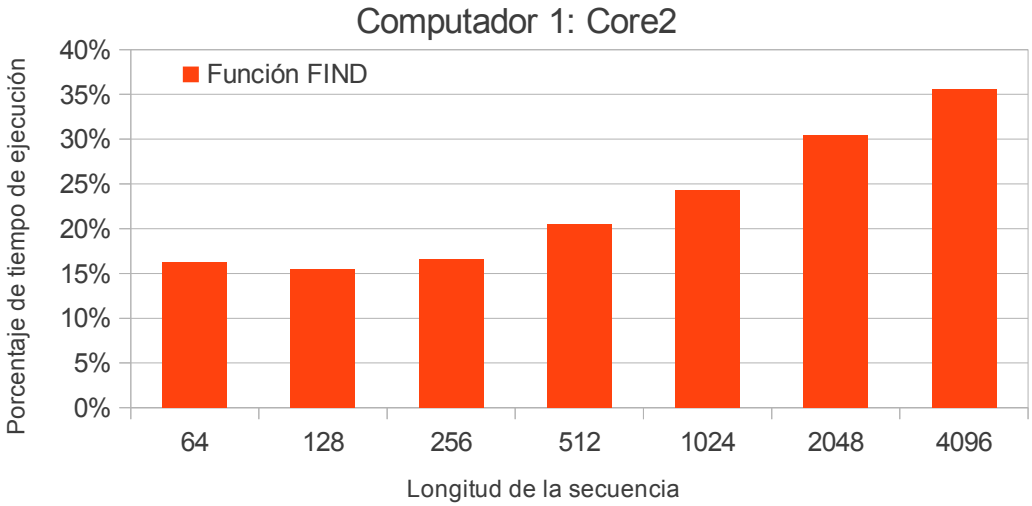


***Fig. 4.2:*** *Porcentaje de tiempo de ejecución de la función 's_BlastAaWordFinder_TwoHit' sobre la ejecución de blastp.*

La función *SCAN* es la encargada de detectar y almacenar los *hits* entre una secuencia de la base de datos y la *query*. Es la función más significativa para *queries* de tamaño pequeño. En cambio, la función *FIND* se encarga de intentar extender los *hits* que la función anterior ha detectado y es la más significativa para *queries* de gran tamaño.

Se puede apreciar cómo sólo la suma de estas dos funciones representan entre el 40% y el 70% de la ejecución total del programa dependiendo de la información proteínica de la *query* aunque sobretodo de su longitud. Además, se puede apreciar que, dependiendo de la longitud de la secuencia, ambas funciones tienen un comportamiento inverso. Si se aumenta la longitud de la *query*, el peso en la aplicación de la función *SCAN* disminuye mientras que el de la función *FIND* aumenta (figura 4.3).

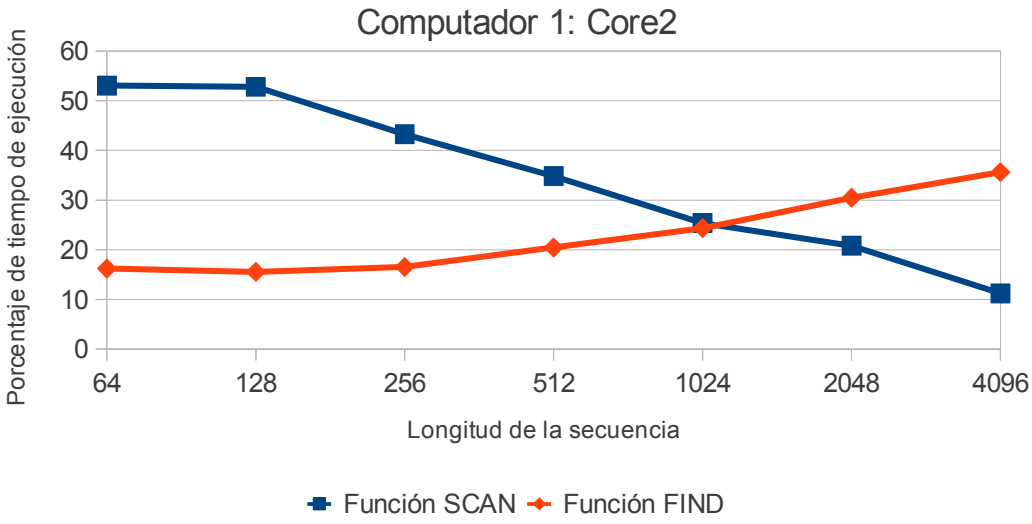


***Fig. 4.3:*** *Comparación de porcentajes de tiempo de ejecución de las funciones 's_BlastSmallAaScanSubject' y 's_BlastAaWordFinder_TwoHit'*

El aumento de longitud proporciona un peso más importante en la ejecución a la función *FIND* mientras que *SCAN* lo pierde, aunque para *queries* pequeñas tiene una gran importancia en el rendimiento.

Ambas funciones contienen un bucle principal que es el que consume la mayor parte del tiempo de ejecución de la función. Para la función *SCAN*, el bucle recorre las *words* (tripletas de aminoácidos) de la secuencia de la base de datos para detectar *hits* con la query de entrada. Para la función *FIND*, el bucle recorre cada *hit* que la función anterior ha detectado.

Como ejemplo, a continuación se muestra la estructura del bucle principal de la función *SCAN* (figura 4.4). Este bucle se ejecuta una vez por cada secuencia almacenada en la *database* y tiene un número de iteraciones igual a N-2, donde N es el número de aminoácidos que contiene cada secuencia. Su funcionalidad es recorrer las tripletas de aminoácidos de la secuencia conocida y comprobar si existe alguna tripleta igual en la secuencia *query*, si se da el caso se procesa un *hit*.

```
for (s = s_first; s <= s_last; s++) {

    index = ComputeTableIndexIncremental(word_length, csize, mask, s, index);

    if (PV_TEST(pv, index, PV_ARRAY_BTS)) {

        /* PROCESAR HIT */
    }
}
```

***Fig. 4.4:*** *Código del bucle principal de la función s_BlastSmallAaScanSubject (SCAN)*

Para intentar optimizar la ejecución de las funciones *SCAN* y *FIND* comentadas anteriormente se han implementado las siguientes optimizaciones:

- Funciones *inline*: el compilador inserta el código completo de una función en cada lugar donde se llama, en vez de generar el código para llamar dicha función. De esta forma podemos evitar llamadas múltiples a funciones de pequeño tamaño.

  Por ejemplo, para el código descrito anteriormente (figura 4.4), se ha declarado la función *ComputeTableIndexIncremental* como *inline* ya que contiene poco código y es llamada en múltiples ocasiones (figura 4.5).

```
for (s = s_first; s <= s_last; s++) {

    index  = ((index << csize) | s[word_length-1]) & mask;

    if (PV_TEST(pv, index, PV_ARRAY_BTS)) {

        /* PROCESAR HIT */
    }
}
```

***Fig. 4.5:*** *Ejemplo de uso de funciones inline*

- Expresiones condicionales: se han substituido los saltos condicionales por expresiones condicionales que son evaluadas en tiempo de ejecución y no provocan ningún salto en el flujo de la aplicación.

  Por ejemplo, para el código descrito anteriormente (figura 4.5), el salto de la sentencia *if* provoca un importante número de fallos de predicción de saltos ya que la condición es difícil de predecir. Estos fallos se pueden solventar utilizando expresiones condicionales como se muestra en el siguiente código (figura 4.6).

  Como solo se deben procesar aquellas tripletas que cumplen la condición de la sentencia *if*, primero podemos almacenar en un vector auxiliar (*aux*) aquellas tripletas que cumplen dicha condición y más tarde se pueden procesar en otro bucle provocando menos fallos de predicción de saltos ya que los saltos de un bucle *for* son más fáciles de predecir.

```
for (s = s_first; s <= s_last; s++) {

    index  = ((index << csize) | s[word_length-1]) & mask;

    aux[j].s = s;
    aux[j].index = index;

    j = (PV_TEST(pv, index, PV_ARRAY_BTS))? j+1 : j;
}

for (k=0; k<j; k++) {

    /* PROCESAR HIT */
}
```

***Fig. 4.6:*** *Ejemplo de uso de expresiones condicionales*

- Loop unrolling: aumenta la velocidad de los bucles al reducir el número de instrucciones que controlan la iteración. De esta forma se minimizan las penalizaciones por saltos y permite reordenar las instrucciones eliminando dependencias de datos. Como desventaja, este tipo de optimización aumenta el tamaño del código fuente y dificulta su lectura.

  Por ejemplo, para el código descrito anteriormente (figura 4.6), ambos bucles *for* pueden ser desenrollados. En el siguiente código podemos ver como se implementa esta optimización en el primer bucle para 2 iteraciones (figura 4.7).

  Hay que tener en cuenta que al no saber el número de iteraciones en tiempo de compilación, debemos implementar el loop unrolling tanto para el caso de que se ejecuta un número de iteraciones par como impar. Además, si se renombran las variables temporales se favorece la ejecución fuera de orden.

```
for (s=s_first; s < s_last; s+=2) {

    index  = ((index2 << csize) | s[word_length-1]) & mask;
    index2 = ((index  << csize) | s[word_length]  ) & mask;

    aux[j].s = s;
    aux[j].index = index;

    j = (PV_TEST(pv, index, PV_ARRAY_BTS))? j+1 : j;

    aux[j].s = s+1;
    aux[j].index = index2;

    j = (PV_TEST(pv, index2, PV_ARRAY_BTS))? j+1 : j;
}

if (s == s_last) {

    index = ((index2 << csize) | s[word_length-1]) & mask;

    aux[j].s = s;
    aux[j].index = index;

    j = (PV_TEST(pv, index, PV_ARRAY_BTS))? j+1 : j;
}
```

***Fig. 4.7:*** *Ejemplo de implementación de loop unroll*

### 4.1.1 Experimentos con las optimizaciones implementadas

A continuación se va a realizar una comparativa del rendimiento de la aplicación original y de la aplicación con las optimizaciones que se han mencionado en el apartado anterior. Para la realización de dicha comparativa se va a usar la misma metodología que en el capítulo anterior (ver apartado 3.2).

Si observamos los tiempos de respuesta de la versión de *blastp* original y la optimizada con *queries* de diferentes longitudes, obtenemos un mejor rendimiento con la versión optimizada (figura 4.8):

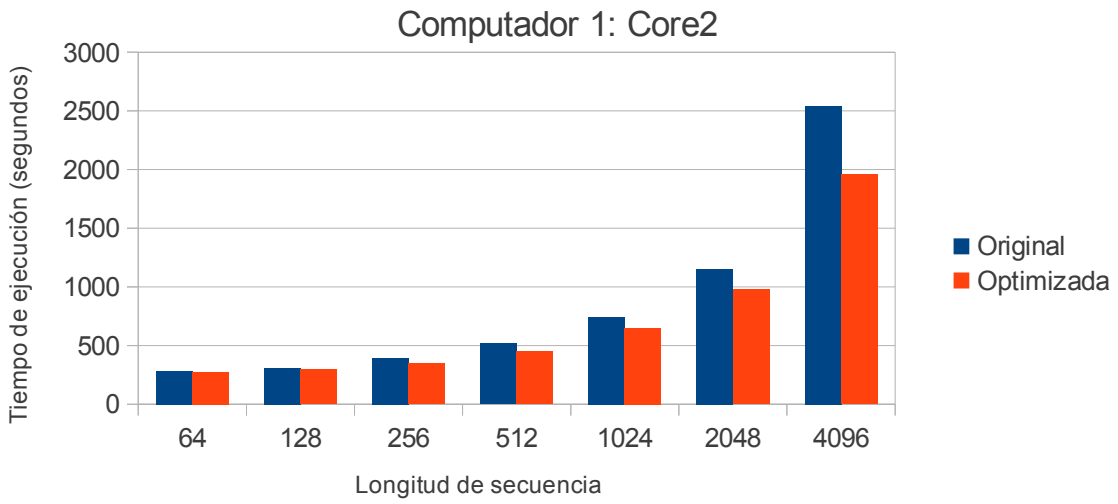


***Fig. 4.8:*** *Tiempos de ejecución (con E/S) de la versión blastp original y de la optimizada*

Si comparamos el número de instrucciones dividido entre la longitud de la secuencia *query* de las dos versiones podemos ver como la versión optimizada reduce el número de instrucciones ejecutadas en un 5-8% (figura 4.9).

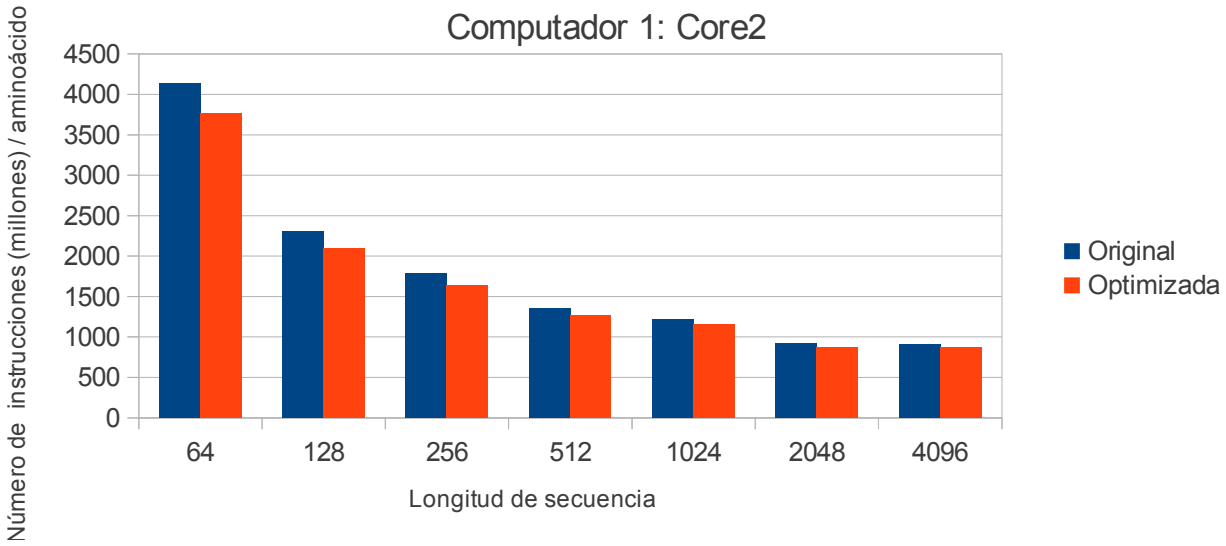


***Fig. 4.9:*** *Número de instrucciones ejecutadas dividido por la longitud de secuencia para la versión de blastp original y optimizada*

Si nos fijamos en la siguiente gráfica (figura 4.10) podemos apreciar como al aumentar la longitud de la *query* también incrementa el speedup:

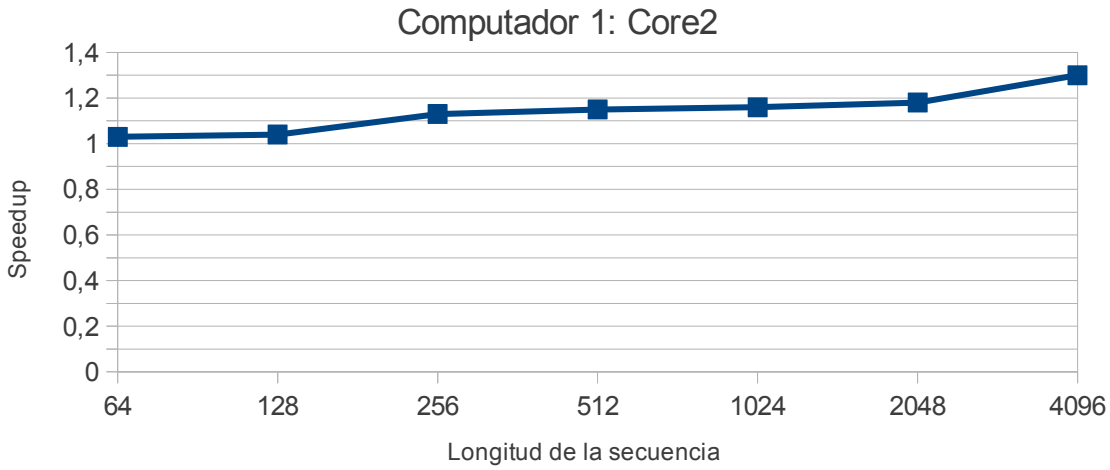


***Fig. 4.10:*** *Speedup de la versión optimizada según la longitud de query*

Este hecho se debe a que al aumentar el tamaño de las *queries* se incrementan también el número de *hits*. Este aumento de *hits* provoca un aumento de iteraciones en el bucle principal de la función s_*BlastAaWordFinder_TwoHit*, que a la vez provoca que las optimizaciones implementadas en dicho bucle tengan cada vez más efecto.

Aunque dichas optimizaciones han sido pensadas para una ejecución secuencial de *blastp*, también se han experimentado en una ejecución multi-core. Como se puede apreciar en la siguiente gráfica (figura 4.11), también se obtiene un rendimiento mejor pero con un speedup no tan bueno:

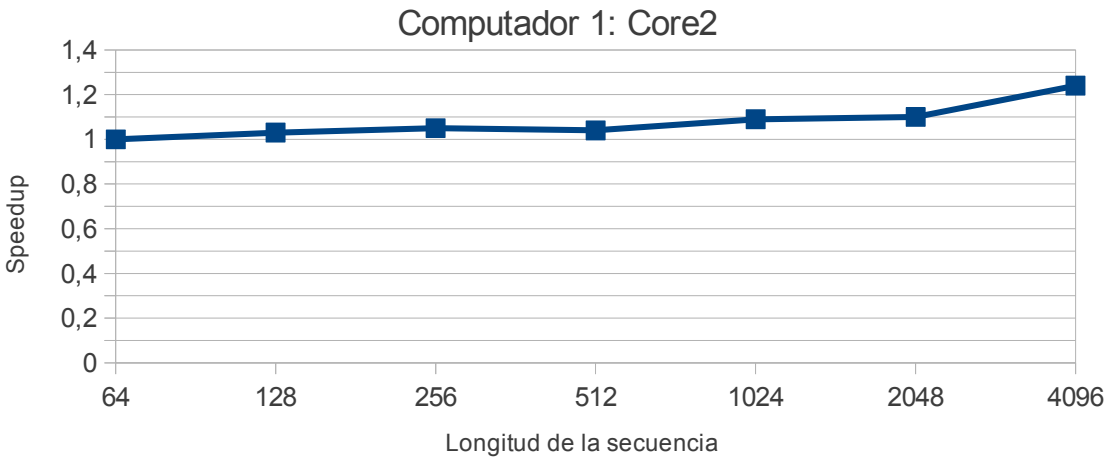


***Fig. 4.11:*** *Speedup de la versión optimizada según la longitud de query y en su versión multi-core (2 threads)*

Se puede entender que al paralelizar la carga de trabajo, el speedup proporcionado por las optimizaciones tiene menos efecto. Sólo se obtiene la mejora del thread que termina último, las optimizaciones de los demás threads no tienen efecto en el tiempo total. Por ejemplo, con una *query* de tan solo 64 aminoácidos se ha obtenido un rendimiento igual a la versión sin optimizar.

## 4.2 Optimizando la aplicación para varias CPUs

En el capítulo anterior, al realizar el análisis de rendimiento de la versión multi-core de *blastp*, se encontraron dos aspectos a destacar que empeoraban el rendimiento: la asignación de los threads en los cores y la asignación de secuencias de la *database* a cada thread.

El primer aspecto, como ya se comentó en el anterior capítulo, se puede solucionar mediante la librería *sched.h*. Mediante la función CPU_SET de dicha librería se puede forzar a cada thread que sea ejecutado en un core distinto, asegurando que dos threads no van a competir por los mismos recursos y mejorando el rendimiento notablemente.

En cuanto a la repartición de las secuencias a procesar para cada thread, se vio en el capítulo anterior que cada thread tenía una carga de trabajo distinta, cosa que provocaba que todos los threads debían esperar al thread más lento para iniciar la fase nº 3. Además, el asignar a cada thread bloques de secuencias de la *database* de gran tamaño, provocaba latencias mayores en la espera a disco causados por movimientos de cabezal del disco. Como distintos threads pedían datos que se encontraban en distintos sectores del disco, esto provocaba que los threads compitieran por el acceso a disco.

Así que para reducir el efecto de este problema se deben repartir las secuencias a cada thread de una forma:

- *Equitativa*: que cada thread procese la misma carga de trabajo, es decir, que procese el mismo número de secuencias.

- *Consecutiva*: que cada thread se reparta de forma consecutiva las secuencias, que es como están almacenadas en disco. De esta forma se evitan posibles movimientos de cabezal innecesarios.

Para conseguir un método que cumpla los criterios anteriores se ha implementado una versión que funciona de la siguiente manera. Si se ejecutan 4 threads, cada thread procesará las secuencias siguientes (figura 4.12):

| Thread | Secuencias procesadas |
|---|---|
| 0 | 1, 5, 9, ..., N-3 |
| 1 | 2, 6, 10, ..., N-2 |
| 2 | 3, 7, 11, ..., N-1 |
| 3 | 4, 8, 12, ..., N |

***Fig. 4.12:*** *Ejemplo de asignación de secuencias a cada thread*

Mediante esta implementación se puede aprovechar que el disco cada vez que trae a memoria los datos pedidos, trae también algunos datos más que son consecutivos. Además, de este modo aseguramos que el número de secuencias procesadas por cada thread va a ser proporcional y equitativa.

### 4.2.1 Experimentos con las mejoras implementadas para multi-core

A continuación podemos ver una comparación del número de ciclos consumidos por cada core en la fase nº 2 con la versión original de *blastp* (figura 4.13) y la versión optimizada usando 2 threads (figura 4.14):

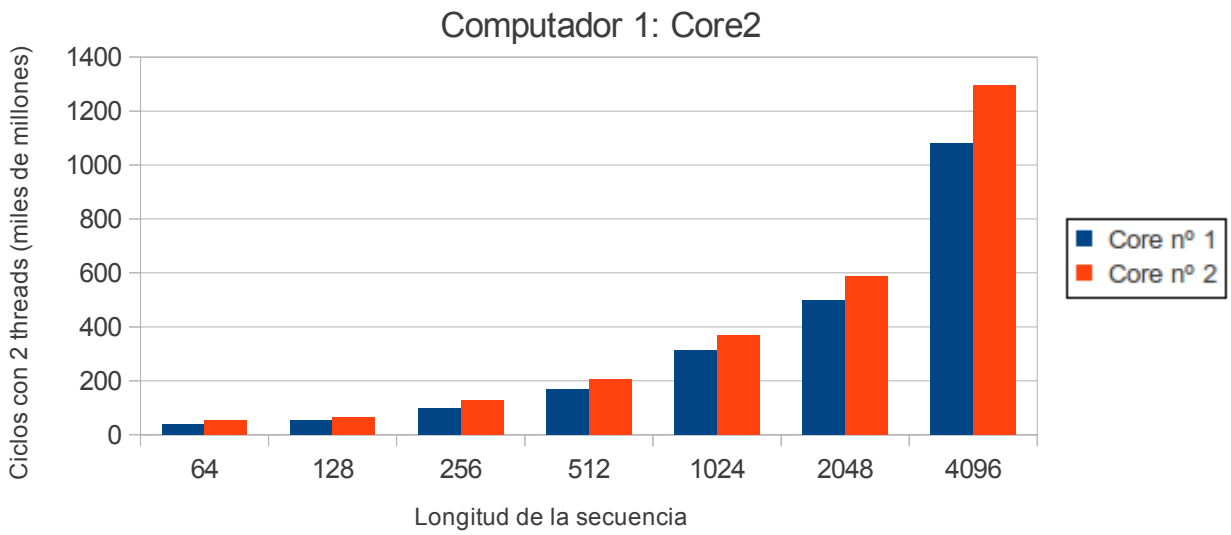


***Fig. 4.13*** *Comparación del número de ciclos consumidos por core de la versión original de blastp en la fase nº 2 (en miles de millones)*

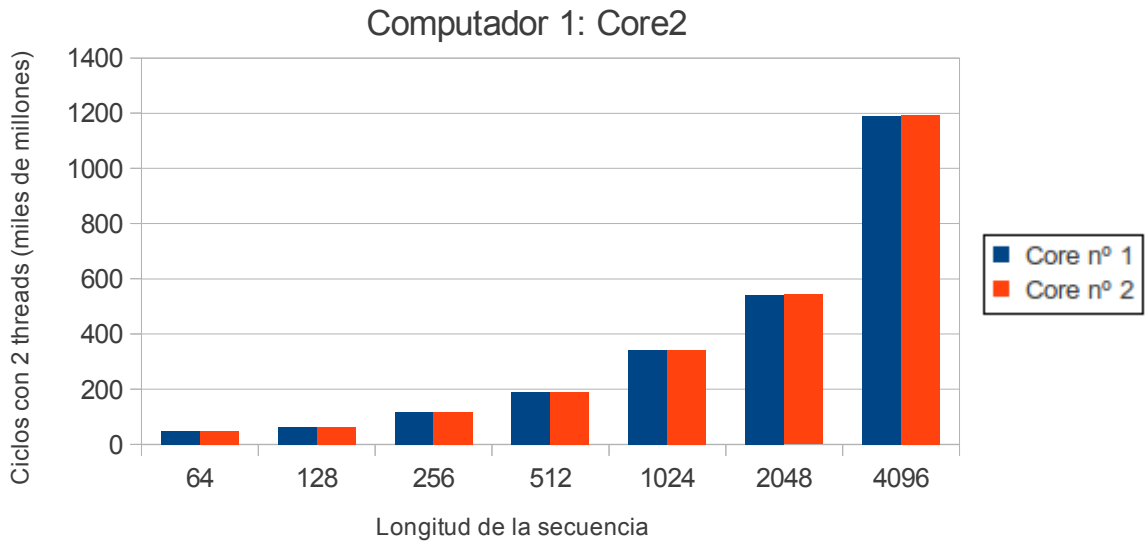


***Fig. 4.14:*** *Comparación del número de ciclos consumidos por core de la versión optimizada de blastp en la fase nº 2 (en miles de millones)*

Podemos ver como para cada secuencia, los ciclos consumidos en cada core por la versión original (figura 4.13) no son equitativos, en cambio los dos cores de la versión optimizada (figura 4.14) tienen una carga de trabajo parecida, que hace disminuir el tiempo de ejecución de la fase.

Mediante esta optimización obtenemos el siguiente speedup (figura 4.15):

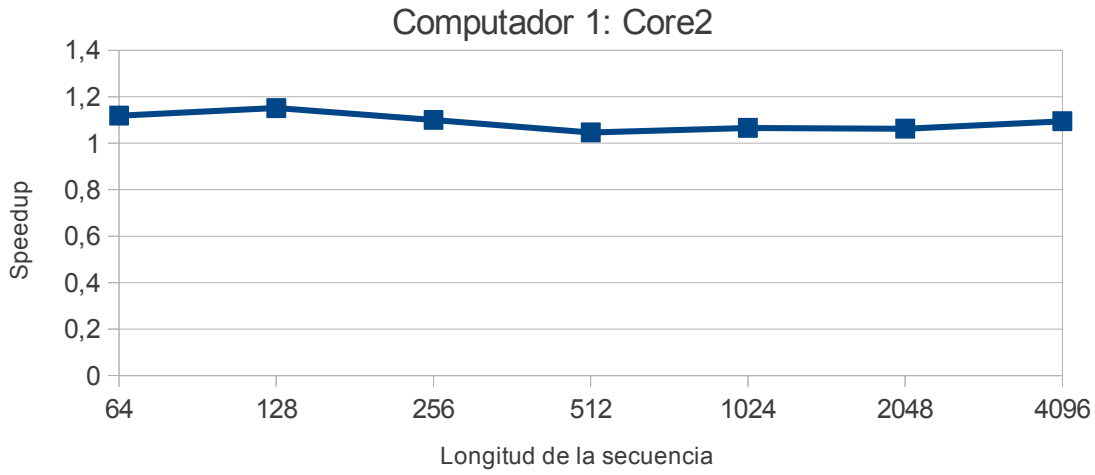


***Fig. 4.15:*** *Speedup de la versión optimizada para varias CPUs*

Se puede apreciar que la mejora es más importante para secuencias pequeñas que para secuencias de gran longitud. Este hecho se debe a que las secuencias pequeñas tienen un tiempo de espera mayor de E/S ya que tienen una carga menor de cómputo que las secuencias de mayor longitud. Por esta razón las secuencias pequeñas obtienen una mejora más importante.

# Capítulo 5

# Conclusiones y líneas futuras

En este capítulo aparecen las conclusiones obtenidas al realizar la investigación. Además, se incluyen alternativas o futuras líneas de investigación que pueden aportar aún más conocimiento al análisis y optimización de *blastp.*

## 5.1 Conclusiones

Cada semana se importan a las *databases* importantes cantidades de secuencias nuevas. Este hecho influye en el manejo de la entrada/salida, que como hemos visto en esta investigación representa uno de los principales problemas en las aplicaciones de alineamiento de secuencias. La gestión de la entrada/salida es el cuello de botella más importante que padece la aplicación, sobretodo con el uso de múltiples cores. Se dispone de muy poca información sobre este manejo, lo que dificulta su estudio para optimizarlo.

Se necesita profundizar más en el tema de la E/S, aunque se ha visto como las propuestas de mejora implementadas afectan notablemente en el rendimiento, el speedup obtenido (una media de 1.09 dependiendo de la secuencia *query*) no es suficiente para tratar *databases* que cada vez adquieren tamaños más grandes.

En cuanto a la parte de procesamiento, también se ha visto como con no muchas modificaciones en el código se han obtenido speedups interesantes (una media de 1.14 dependiendo de la secuencia *query*), aunque el CPI sigue siendo demasiado elevado (una media de 0.8 ciclos por instrucción) y se debería estudiar con más detenimiento su causa.

Además, el aumento de tamaño de las *databases* provocan que herramientas como *blastp* tengan que ser modificadas periódicamente. Las herramientas BLAST han sufrido desde su lanzamiento diferentes mejoras y añadidos que han provocado que su código fuente sea extremadamente grande y complicado. La primera versión se lanzó en el año 1990, pero la versión actual escrita está compuesta por una serie de modificaciones y nuevas características añadidas a la versión del año 1997 que se diseño en lenguaje C. Por esta razón, *blastp* está diseñada en C y C++.

Como se ha podido ver en este proyecto, hay características que se pueden implementar más eficazmente y que afectan muy positivamente en el rendimiento. Pero hay otras optimizaciones que son muy difícil o ni siquiera se pueden implementar por la complejidad que ha sufrido el código por causa de estas actualizaciones o modificaciones.

En conclusión, la aplicación *blastp* debe mejorar en dos aspectos que tras la investigación han sido detectados como potencialmente problemáticos:

- La correcta gestión de la entrada/salida, sobretodo por el importante aumento de tamaño que padecen las bases de datos.

- El correcto balanceo de la carga de trabajo en los cores al usar la versión multi-core.

## 5.2 Líneas futuras

Quedan algunas líneas de investigación abiertas para aportar más conocimiento al análisis de rendimiento o bien para la optimización de *blastp.* A continuación se nombran algunas de ellas:

- Utilización de sistemas Multi-threading

Como se ha visto en este documento, las aplicaciones BLAST presentan problemas de importantes latencias (a memoria, fallos de predicción de saltos, etc). Mediante experimentos con multi-threading se podrían comparar los rendimientos de *blastp* usando el doble de threads que cores dispone la computadora. De este modo se podría estudiar el grado de saturación de los recursos que padece la computadora.

- Implementación con CUDA (*Compute Unified Device Architecture*)

Mediante esta tecnología se podría dividir el cómputo de la fase nº 2 de *blastp* en una gran cantidad de threads, con lo que se paralelizaría mucho más la carga de trabajo obteniendo rendimientos y speedups interesantes. Quizás se caería otra vez en el problema de la lectura de secuencias de las *database* de gran tamaño, que se ha demostrado como un grave problema de eficiencia. Aunque no dejaría de ser una solución a tener en cuenta.

- OpenMP

OpenMP es una interfaz de programación de aplicaciones para la programación multiproceso de memoria compartida. Permite añadir concurrencia a los programas escritos en C y C++ como *blastp,* usando el modelo de ejecución fork-join.

Al tratarse de un modelo de programación modulable y escalable que proporciona a los programadores una interfaz simple y flexible para el desarrollo de aplicaciones paralelas, se puede usar para rediseñar *blastp*, que después de diversas modificaciones en su código se ha convertido en una aplicación compleja. Por esta razón se podría decir que para optimizar de una forma mucho más eficiente *blastp* se tendría que realizar un nuevo diseño, pudiendo aprovechar las ideas ya implementadas en las versiones actuales pero reescribiendo el código de nuevo usando OpenMP.

# Bibliografía


- ALTSCHUL, GISH, MILLER, MYERS and LIPMAN. “Basic Local Alignment Search Tool”. Journal of Molecular *Biology*.1990.
- CHAO, Kun-Mao y ZHANG, Louxin. “Sequence Comparison: Theory and Methods”. Springer. 2009.
- CHURBANOV, A. “Pam matrix for Blast algorithm,” University of Nebraska. 2002.
- Desconocido. “BLAST Help Manual for Unix”. National Center for Biotechnology Information (NCBI). 2008.
- Desconocido. “The GenBank Submissions Handbook”. National Center for Biotechnology Information (NCBI). 2011.
- KOONIEN, Eugene y Galperin, Michael. “Computational Approaches in Comparative Genomics”. Boston: Kluwer Academic. 2003.
- LOUKIDES, Mike y MASUMECI, Gian-Paolo. “System performance Tuning”. O'Reilly & Associates. 2002.
- MC ENTYRE, Jo y Ostell, Jim. “The NCBI Handbook”. National Center for Biotechnology Information (NCBI). 2002.
- MOUNT, David. “Bioinformatics: Sequence and Genome Analysis”. Cold Spring Harbor Laboratory Press. 2001.
- MÜLLER, Matthias. “Tools for high performance computing”. Springer. 2009.
- POLANSKI, Andrzej y KIMMEL, Marek. “Bioinformatics”. Springer. 2007.
- SANDERS, Jason y KANDROT, Edward. “CUDA by Example: An Introduction to General-Purpose GPU Programming”. Addison-Wesley Professional. 2010.
- SCHMIDT, Bertil. “Bioinformatics: High Performance Parallel Computer Architectures”. CRC Press. 2010.
- VAKATOV, Denis. “The NCBI C++ Toolkit Book”. National Center for Biotechnology Information (NCBI). 2004.

Signatura de l'alumne:

Carles Figuera Penedo

Bellaterra, 21 de Juny de 2011

## Resumen

Las aplicaciones de alineamiento de secuencias son una herramienta importante para la comunidad científica. Estas aplicaciones bioinformáticas son usadas en muchos campos distintos como pueden ser la medicina, la biología, la farmacología, la genética, etc. A día de hoy los algoritmos de alineamiento de secuencias tienen una complejidad elevada y cada día tienen que manejar un volumen de datos más grande. Por esta razón se deben buscar alternativas para que estas aplicaciones sean capaces de manejar el aumento de tamaño que los bancos de secuencias están sufriendo día a día. En este proyecto se estudian y se investigan mejoras en este tipo de aplicaciones como puede ser el uso de sistemas paralelos que pueden mejorar el rendimiento notablemente.

## Resum

Les aplicacions d'alineament de seqüències són una eina important per a la comunitat científica. Aquestes aplicacions bioinformàtiques són utilitzades en molts camps diferents com poden ser la medicina, la biologia, la farmacologia, la genètica, etc. A dia d'avui els algorismes d'alineament de seqüències tenen una complexitat elevada i cada dia han de gestionar un volum de dades més gran. Per això s'han de buscar alternatives per a que aquestes aplicacions siguin capaces de gestionar l'augment de mida que els bancs de seqüències estan patint dia a dia. En aquest projecte s'estudien i s'investiguen millores en aquest tipus d'aplicacions com pot ser l'ús de sistemes paral·leles que poden millorar el rendiment notablement.

## Abstract

The sequence alignment applications are an important tool for the scientific community. These bioinformatics applications are used in many different fields such as medicine, biology, pharmacology, genetics, etc. Today the sequence alignment algorithms are highly complex and every day have to handle a large volume of data. For this reason we must find alternatives for these applications are able to handle the increased size of sequences that banks are suffering every day. In this project we study and investigate improvements in these applications such as the use of parallel systems that can improve performance significantly.