Alineamiento de Secuencias Genéticas en Procesadores MultiCore

Memòria del Projecte Fi de Carrera

d'Enginyeria en Informàtica realitzat per

Carles Pons Noguera

i dirigit per

Juan Carlos Moure López

Bellaterra, 22 de Juny de 2010

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 Pons Noguera

I per tal que consti firma la present.

Signat: .....

Bellaterra, 22 de Juny de 2010

ÍNDICE

1. Introducción

La evolución de los procesadores ha pasado por diversas etapas en el transcurso de los años, con el objetivo de aumentar el rendimiento posible. La primera etapa por la que pasó fue el aumento del número de transistores que podían caber en un chip de silicio, doblándose esta cifra cada 18 meses [1]. En la segunda etapa el rendimiento siguió incrementándose a costa del aumento la frecuencia de reloj del procesador.

Tanto con el aumento del número de transistores como con el aumento de la frecuencia del reloj se vió que los transistores generaban demasiado calor, y un procesador caliente puede hacer que un ordenador falle. Los ordenadores con procesadores rápidos necesitaban un sistema de refrigerado eficiente para evitar el sobrecalentamiento. Cuanto más se aumentaba el tamaño del chip del procesador y/o la frecuencia de reloj, más calor generaba el equipo cuando trabajaba a toda velocidad y mayor era el coste económico dirigido a disipar el calor que producía.

La alternativa propuesta por la industria para aumentar la velocidad de ejecución sin aumentar el número de transistores ni la frecuencia de reloj fueron los procesadores multicore. Los procesadores multicore combinan 2 o más procesadores en un mismo chip. Se trata de aumentar el rendimiento mediante el paralelismo .
Con la aparición de los procesadores multicore las aplicaciones singlethread no pueden aprovecharse del rendimiento extra, se requiere paralelizar las aplicaciones para aumentar la velocidad de ejecución de las aplicaciones.
Este cambio en la programación de aplicaciones dificulta el trabajo del programador. Por lo que el programador necesitará de una metodología para el diseño e implementación de aplicaciones paralelas. Una forma de dividir las instrucciones entre los procesadores es el uso de la API OpenMP.

1.1. Objetivos

Este trabajo analiza el tiempo de ejecución y el rendimiento de una aplicación en sistemas multicore y multithreading. El objetivo es aumentar el rendimiento de la aplicación y disminuir lo máximo posible su tiempo de ejecución.

La aplicación que se estudiará pertenece al campo de la Bioinformática, es el algoritmo de alineamiento de secuencias genéticas denominado Needleman-Wunsch.
El alineamiento de secuencias es una forma de comparar dos o más secuencias o cadenas de ADN, ARN, o proteínas para resaltar sus zonas de similitud.

Los sistemas donde analizaremos las aplicaciones tienen en común que son multicore, es decir, están formados por varios cores. También son multithreading, pueden ejecutar diversos threads en cada core.

Objetivos:

- Aprender el funcionamiento de los algoritmos de comparación de secuencias genéticas.

- Codificar el algoritmo serie, plantear una serie de optimizaciones y analizar el rendimiento.

- Diseñar y implementar la versión en paralelo del algoritmo, así como intentar planificar la correcta distribución de los threads para ejecutar en procesadores multicore.

- Medir las prestaciones del algoritmo con el que vamos a trabajar, y comparar entre los diferentes tipos de procesadores multicore y multithread

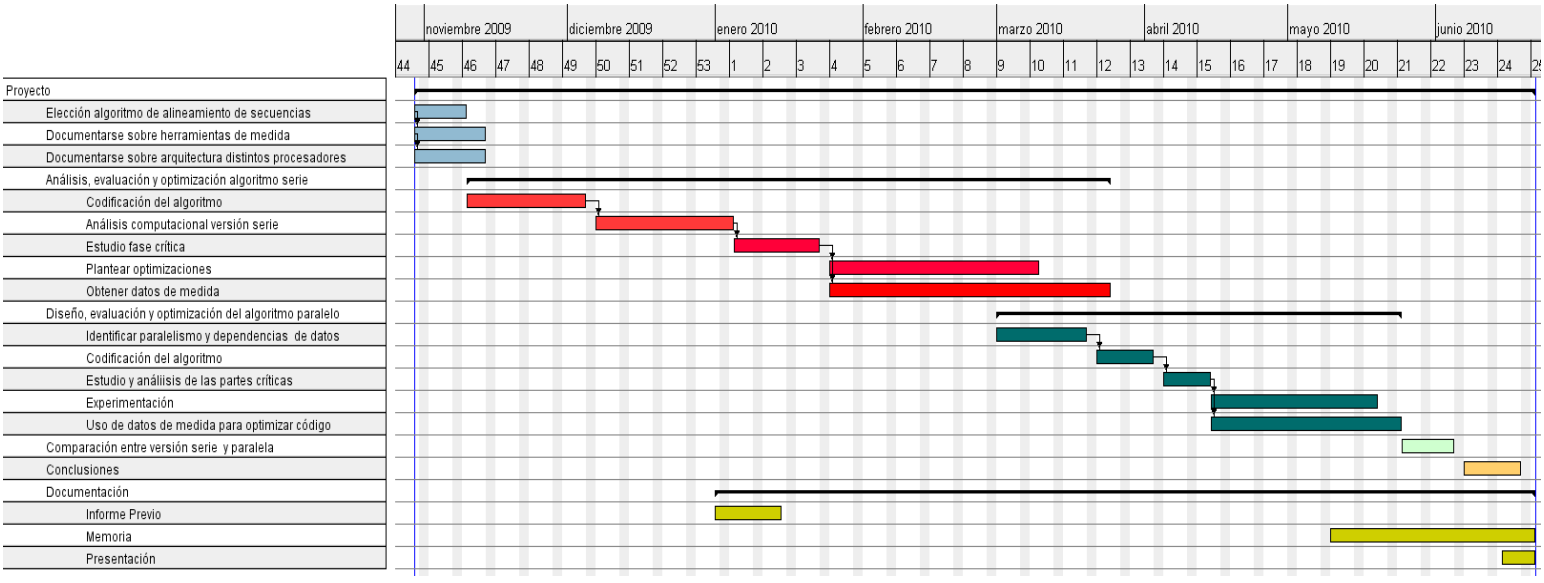
- Realizar una investigación de todos los resultados de tiempo de ejecución de la aplicación, e utilizar los datos de medida de prestaciones para optimizar el código, mejorar el tiempo de ejecución, el rendimiento y el uso de memoria.

1.2. Planificación temporal del trabajo

Las tareas a realizar serán :

- Elección del algoritmo de alineamiento de secuencias.
- Documentarse sobre herramientas de medida.
- Documentarse sobre la arquitectura de distintos procesadores.
- Análisis, evaluación y optimización del algoritmo serie.
    - Codificación del algoritmo.
    - Análisis computacional de la versión serie.
    - Estudio de la fase crítica.
    - Plantear optimizaciones.
    - Obtener datos de medida.
- Diseño, evaluación y optimización del algoritmo paralelo.
    - Identificar paralelismo y dependencias de datos.
    - Codificación del algoritmo.
    - Estudio y análisis de las partes críticas.
    - Experimentación.
    - Uso de datos de medida para optimizar código.
- Comparación entre los algoritmos serie y paralelo.
- Conclusiones.
- Realización de la documentación.

El diagrama de Gantt posterior representa la planificación temporal del trabajo con las tareas que hemos definido previamente.



1.3. Esquema de la memoria

La memoria se compone de 7 capítulos, que serán descritos brevemente a continuación.

Capítulo 1: Se describen los objetivos del trabajo, la planificación de trabajo y finalmente se describen los diferentes capítulos de los que consta la memoria.

Capítulo 2: Marco teórico, en el que se describen y explican los conceptos necesarios que engloban este trabajo. Se introduce al lector en conceptos relacionados con la evolución de los procesadores, las limitaciones de los procesadores singlethread, los procesadores multicore/multithread, los factores que limitan el rendimiento en sistemas multicore/multithread y OpenMP.

Capítulo 3: En él se plantea y describe el algoritmo de Needleman-Wunsch así como su ubicación dentro del mundo de la biología.

Capítulo 4: Se realiza el diseño y estudio del algoritmo serie, así como las posibles mejoras que se han ido aplicando a éste. Acto seguido, se exponen los resultados (tiempo de ejecución) para diferentes arquitecturas de computadores. Finalmente, se explican e interpretan los resultados para ambos sistemas.

Capítulo 5: En este capítulo se discute la posibles maneras de paralelizar el algoritmo, se realiza el diseño y estudio del algoritmo paralelo de la variante de grano fino y de grano grueso. Finalmente se explican y comparan los resultados de ambas variantes.

Capítulo 6: Incluye las conclusiones y las líneas futuras de investigación.

Capitulo 7: Bibliografia

2. Marco Teórico

En este capítulo describiremos el procesador clásico así como sus limitaciones, para explicar posteriormente los procesadores multithread y multicore, y los problemas que impiden conseguir un alto rendimiento en estos sistemas. Finalmente realizaremos una breve introducción a OpenMP.

2.1. Evolución del procesador singlethread

- Procesador clásico

El procesador clásico ejecuta instrucciones almacenadas como números binarios organizados secuencialmente en la memoria principal. La ejecución de las instrucciones se realiza en varias fases:

- PreFetch, pre lectura de la instrucción desde la memoria principal.
- Fetch, envío de la instrucción al decodificador
- Decodificación de la instrucción, es decir, determinar qué instrucción es y por tanto qué se debe hacer.
- Lectura de operandos (si los hay).
- Ejecución, lanzamiento de las máquinas de estado que llevan a cabo el procesamiento.
- Escritura de los resultados en la memoria principal o en los registros.

Cada una de estas fases se realiza en uno o varios ciclos de CPU, dependiendo de la estructura del procesador. La duración de estos ciclos viene determinada por la frecuencia de reloj, y nunca podrá ser inferior al tiempo requerido para realizar la tarea individual (realizada en un solo ciclo) de mayor coste temporal. Además hasta que una instrucción no finaliza no se puede comenzar la búsqueda de la siguiente instrucción.

- Procesador Segmentado

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.

La segmentación es una técnica de implementación de procesadores que desarrolla el paralelismo a nivel de intrainstrucción. Mediante la segmentación se puede solapar la ejecución de múltiples instrucciones. El procesamiento segmentado aprovecha la misma filosofía de trabajo de la fabricación en cadena : cada etapa de la segmentación completa una parte de la tarea total. Los segmentos están conectados cada uno con el siguiente, de forma que la salida de uno pasa a ser la entrada del siguiente. Así, los segmentos configuran un cauce a través del que se va procesando la tarea deseada . Lo mas interesante de la segmentación es que las diferentes subtareas pueden procesarse de forma simultánea, aunque sea sobre diferentes datos.

A diferencia del procesador clásico, la segmentación nos ofrece la posibilidad de comenzar una nueva tarea sin necesidad de que la anterior se haya terminado. La medida de eficacia de un procesador segmentado no es el tiempo total transcurrido desde que se comienza una determinada tarea hasta que se termina (tiempo de latencia del procesador), sino el tiempo máximo que puede pasar entre la finalización de dos tareas consecutivas.

Evidentemente, la segmentación no es posible sin incrementar los recursos de proceso.

- Procesador superescalar

Un procesador superescalar es aquel que es capaz de procesar más de una instrucción simultáneamente en cada una de las etapas De esta manera pueden aumentar el paralelismo a nivel de instrucción.
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 :

- Estructurales: cuando dos instrucciones requieren el mismo tipo de unidad funcional pero su número no es suficiente.
- De datos: cuando una instrucción necesita el resultado de otra para ejecutarse.
- De escritura: cuando dos instrucciones necesitan escribir en el mismo registro de memoria.

Podemos distinguir diferentes tipos de procesadores por la forma de actuar ante una dependencia. En un procesador con ejecución en orden las instrucciones quedarán paradas a la espera de que se resuelva la dependencia. Mientras que en un 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 [2].

- Limitaciones del procesador singlethread

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 llevar hasta 500 ciclos de reloj en los que el procesador debe esperar hasta que el acceso a memoria se haya completado *(Figura 1)*.

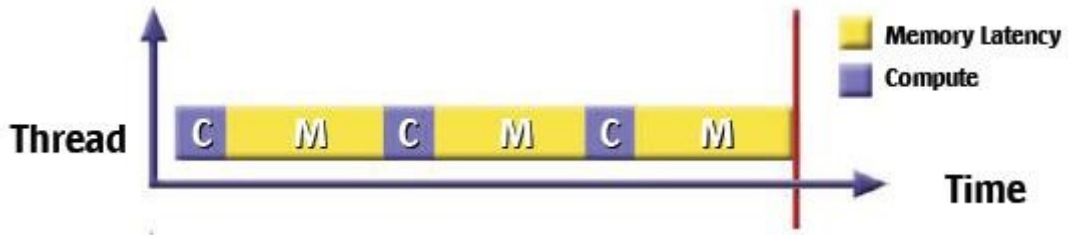


Figura 1 : Fases de la ejecución de un programa singlethread

Por tanto, un aumento de la frecuencia 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. Por ejemplo Una disminución del 50% del tiempo de CPU supone menos de un 10% de mejora en tiempo total de computación *(Figura 2)*.

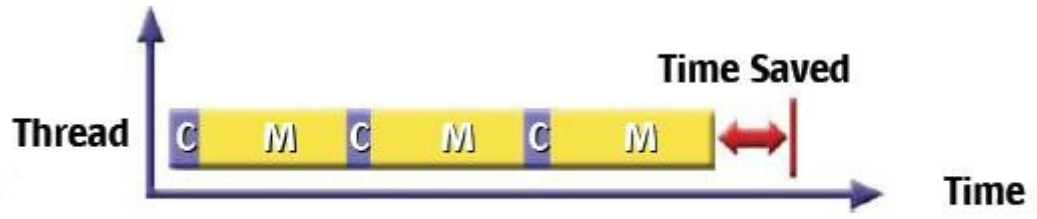


Figura 2 : Fases de la ejecución de un programa singlethread con un procesador más rápido.

La solución para aprovechar todos estos ciclos que se pierden en cada acceso a memoria es el multi-threading por hardware.

El multithreading 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 *(Figura 3)*. Esta solución se analizará en el siguiente apartado.

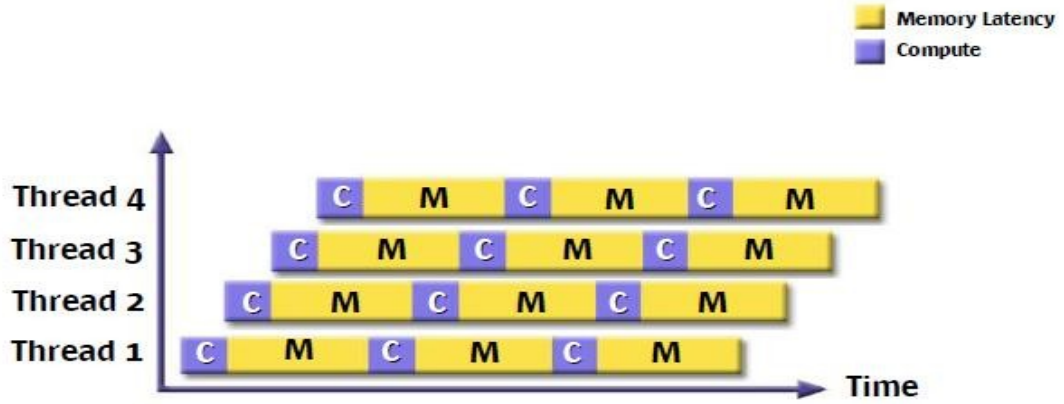


Figura 3 : Fases de la ejecución de un programa usando multithreading.

El incremento de la frecuencia de reloj del procesador es otro de los problemas del procesador singlethread ya que este incremento 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, de la pequeña 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.

2.2. Procesador multithread

Un thread, en sistemas operativos, es una característica que permite a una aplicación realizar varias tareas concurrentemente. Los distintos threads comparten una serie de recursos tales como el espacio de memoria, la pila de ejecución, el estado de la CPU, etc. Esta técnica permite simplificar el diseño de una aplicación que debe llevar a cabo distintas funciones simultáneamente.

En ocasiones no es posible explotar el paralelismo a nivel de instrucción dentro de un proceso debido a las dependencias que existen entre unas instrucciones y otras y que impiden su ejecución paralela. Una posible solución es el multithreading, que permite explotar el paralelismo entre instrucciones que pertenecen a diferentes hilos de ejecución o threads, y que por lo tanto es menos probable que dependan unas de otras (TLP o paralelismo a nivel de thread).

El multihtreading consiste en ejecutar al mismo tiempo dos o más threads de un programa, permitiendo que cada uno de estos threads sea planificado de la manera más conveniente en el procesador, es decir, aprovechando al máximo todos los recursos disponibles. Es equivalente a tener dos o más procesadores lógicos o virtuales en lugar de uno sólo. [3]

Existen diferentes políticas de multithreading, que son :

- Large-grain multithreading :

El procesador ejecuta el thread 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 thread *(Figura 4)*. Este método tiene la ventaja de ser sencillo de implementar.

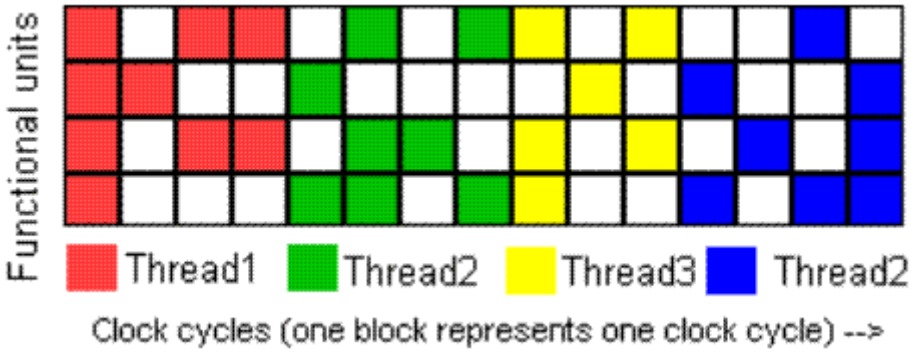


Figura 4: Ejecución de threads siguiendo el modelo Large-grained Multithreading.

- Fine-grain multithreading:

Se basa en un cambio rápido entre threads, ejecutando en cada ciclo un thread 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 threads bloqueados, se ejecutan instrucciones de diferentes threads. Este enfoque tiene la ventaja de eliminar las dependencias de datos que paran el procesador. Al pertenecer las instrucciones a diferentes threads, las dependencias de datos desaparecen *(Figura 5)*.

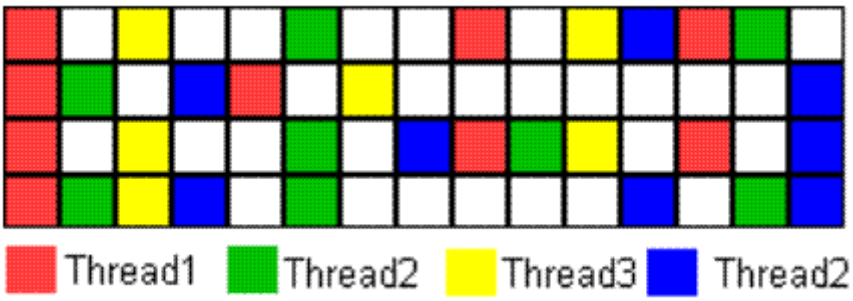


Figura 5 : Ejecución de threads siguiendo el modelo Fine-grained Multithreading.

- Simultaneous multithreading :

Consiste en permitir que se emitan en el mismo ciclo instrucciones que pertenecen a distintos hilos de ejecución o threads *(Figura 6)*. El mecanismo se ha creado utilizando como base un procesador superescalar con planificación dinámica fuera de orden. Desarrollar el SMT requiere un hardware adicional para toda la lógica.

Figura 6 : Ejecución de threads siguiendo el modelo Simultaneous Multithreading.

2.3. Multicore

Los procesadores multicore combinan dos o más procesadores independientes en un mismo chip. Los procesadores se pueden configurar para que ejecute cada uno una parte de un programa o varios programas al mismo tiempo. Un diagrama de bloques de esta arquitectura se muestra en la figura 7. Un multiprocesador está generalmente formado por *n* procesadores y *m* módulos de memoria. La red de interconexión conecta cada procesador a un subconjunto de los módulos de memoria.

Dado que los multiprocesadores comparten los diferentes módulos de memoria, pudiendo acceder varios procesadores a un mismo módulo, a los multiprocesadores también se les llama *sistemas de memoria compartida*. Dependiendo de la forma en que los procesadores comparten la memoria, podemos hacer una subdivisión de los multiprocesadores:

- UMA (*Uniform Memory Access*) :

En un modelo de *Memoria de Acceso Uniforme*, la memoria física está uniformemente compartida por todos los procesadores. Esto quiere decir que todos los procesadores tienen el mismo tiempo de acceso a todas las palabras de memoria. Cada procesador puede tener su cache privada, y los periféricos son también compartidos de alguna manera [4]. La figura 7 muestra el modelo UMA de un multiprocesador.
Es frecuente encontrar arquitecturas de acceso uniforme que además tienen coherencia de caché, a estos sistemas se les suele llamar CC-UMA (*Cache-Coherent Uniform Memory Access*).

A los procesadores los llamamos P1, P2, . . . , Pn y a las memorias M1,M2, . . . ,Mn

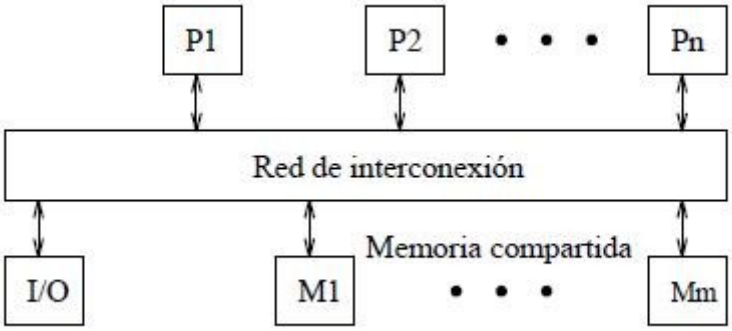


Figura 7 : Modelo de arquitectura UMA de multiprocesador

- NUMA (*Non Uniform Memory Access*) :

Un multiprocesador de tipo NUMA es un sistema de memoria compartida donde el tiempo de acceso varía según el lugar donde se encuentre localizado el acceso. La figura *8* muestra una posible configuración de tipo NUMA, donde toda la memoria es compartida pero local a cada módulo procesador. Otras posibles configuraciones incluyen los sistemas basados en agrupaciones (*clusters*) de sistemas como el de la figura que se comunican a través de otra red de comunicación que puede incluir una memoria compartida global. [4]

La ventaja de estos sistemas es que el acceso a la memoria local es más rápido que en los UMA aunque un acceso a memoria no local es más lento. Lo que se intenta es que la memoria utilizada por los procesos que ejecuta cada procesador, se encuentre en la memoria de dicho procesador para que los accesos sean lo más locales posible.

Aparte de esto, se puede añadir al sistema una memoria de acceso global. En este caso se dan tres posibles patrones de acceso. El más rápido es el acceso a memoria local. Le sigue el acceso a memoria global. El más lento es el acceso a la memoria del resto de módulos.

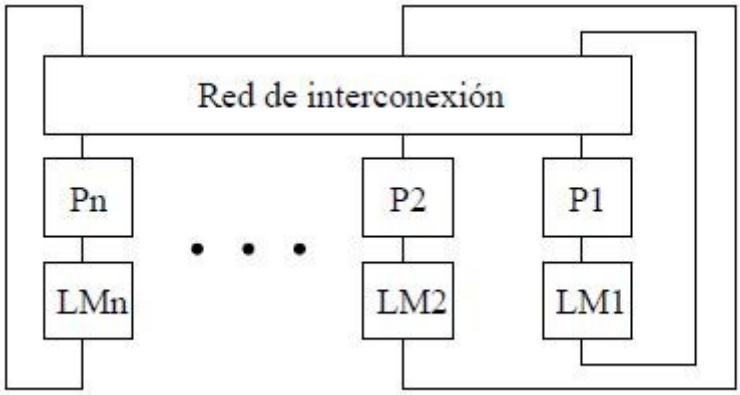


Figura 8 : Modelo de arquitectura NUMA de multiprocesador

2.4. Evaluación del rendimiento de un programa

Para obtener un alto rendimiento del sistema es necesario que haya una sintonía entre la capacidad de la máquina y el comportamiento del programa.
La capacidad de la máquina es susceptible de mejora con las nuevas tecnologías hardware y software, además de la gestión eficiente de los recursos.

El comportamiento del programa depende básicamente de los siguiente factores:

- Diseño del algoritmo
- Estructuras de datos
- Eficiencia de los lenguajes
- Conocimientos del programador
- Tecnología de los compiladores

Para la evaluación del rendimiento de un programa se utilizan una serie de parámetros que conforma un modelo simplificado de la medida del rendimiento de un sistema. Dentro de este modelo, estos son los indicadores de rendimiento más utilizados :

- Frecuencia de reloj (f) : Es la inversa del tiempo de ciclo. f = 1/. Medida en Megahertz.

- Total de Instrucciones (Icount) : Es el número de instrucciones objeto a ejecutar en un programa.

- Ciclos por instrucción (CPI) : Es el número de ciclos que requiere cada instrucción.

- Tiempo de ejecución de programa (Tp) Es el tiempo que tarda un programa en ejecutarse.
  $Tp = Ic * CPI * = Ic * CPI/f = C/f$

- Total de ciclos de reloj en la ejecución de un programa (C) : $C = Ic * CPI$

2.5. Factores que impiden un alto rendimiento en sistemas multicore y multithread

Si el rendimiento de las aplicaciones en sistemas multicore/multithread escalara linealmente al aumentar el número de threads con los que se ejecuta la aplicación, el problema estaría resuelto. Pero lo cierto es que esta escena no se produce nunca ya que existen una serie de factores que impiden que esta escalabilidad sea lineal, estos son:

- Overheads por creación/eliminación de threads

La creación y destrucción de los threads que trabajan en paralelo, tiene un coste en tiempo (overhead). La importancia de este overhead dependerá de la relación entre el (tiempo total de ejecución del bucle con un thread / nº de threads) y (tiempo que se tarda en crear los threads, repartir el trabajo, recoger el resultado y eliminar los threads).El overhead ha de ser pequeño en comparación con (tiempo de ejecución / n threads) sino no nos resulta rentable trabajar con threads.

- Desbalanceo de carga

Una incorrecta distribución del volumen de cómputo por thread implicará que algunos threads finalicen su trabajo antes que otros. Por lo que los threads que han finalizado tendrán que esperar. 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 threads.

- Las comunicaciones entre las memorias de los cores

Los threads de una ejecución multithread pueden trabajar de manera independiente y con datos independientes. Pero por las características de las aplicaciones, en algún momento necesitarán intercambiar datos. Este intercambio de datos se realiza de manera transparente al thread ya que éste únicamente accederá a unas posiciones de memoria que previamente otro thread habrá modificado. Esto aunque es transparente para el thread no esta libre de coste en tiempo. Los datos que hayan sido modificados en la cache de un thread tendrán que ser copiados a la cache del thread que los necesita en ese momento.
Hay que tener en cuenta que el coste de comunicar datos modificados por threads que se ejecutan dentro de un mismo procesador es muy inferior al coste de comunicar datos entre threads que se ejecutan en cores de diferentes procesadores.

Estos costes suponen un overhead a considerar a la hora de diseñar una aplicación multithread. Habrá que prestar especial atención a la localidad temporal y espacial de la aplicación y a la descomposición por dominio, intentando primero minimizar las comunicaciones entre threads que se ejecutan en distintos procesadores y segundo minimizar las comunicaciones entre threads que se ejecutan en el mismo procesador.

2.6. OpenMP

OpenMP es una API para la programación multiproceso de memoria compartida en múltiples plataformas. Permite añadir concurrencia a los programas escritos en C, C++ y Fortran sobre la base del modelo de ejecución fork-join. Está disponible en muchas arquitecturas, incluidas las plataformas de Unix y de Microsoft Windows.

OpenMP se basa en el modelo fork-join *(Figura 9)*, paradigma que proviene de los sistemas Unix, donde una tarea muy pesada se divide en K hilos (fork) con menor peso, para luego "recolectar" sus resultados al final y unirlos en un solo resultado (join).

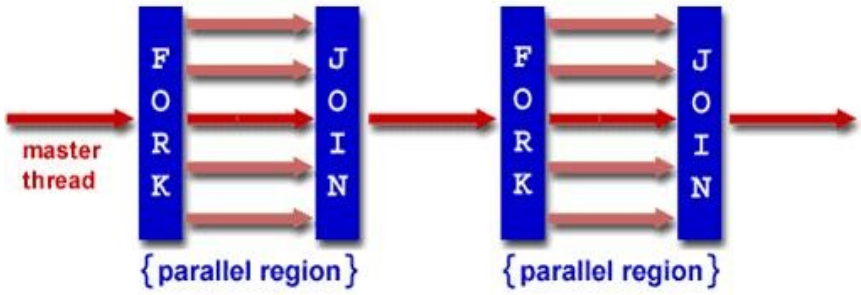


Figura 9: Modelo fork-join

Cuando se incluye una directiva OpenMP esto implica que se incluye una sincronización obligatoria en todo el bloque. Es decir, el bloque de código se marcará como paralelo y se lanzarán threads según las características que nos dé la directiva, y al final de ella habrá una barrera para la sincronización de los diferentes hilos (salvo que implícitamente se indique lo contrario con la directiva nowait). Este tipo de ejecución es la que llamamos **fork-join**. [5]

La sintaxis básica que nos encontramos en una directiva de OpenMP es:

```
# pragma omp <directiva> [cláusula [ , ...] ...]
```

Directivas a destacar

- **parallel**: Indica la parte del código que se podrá ejecutar por varios threads *(Figura 10)*

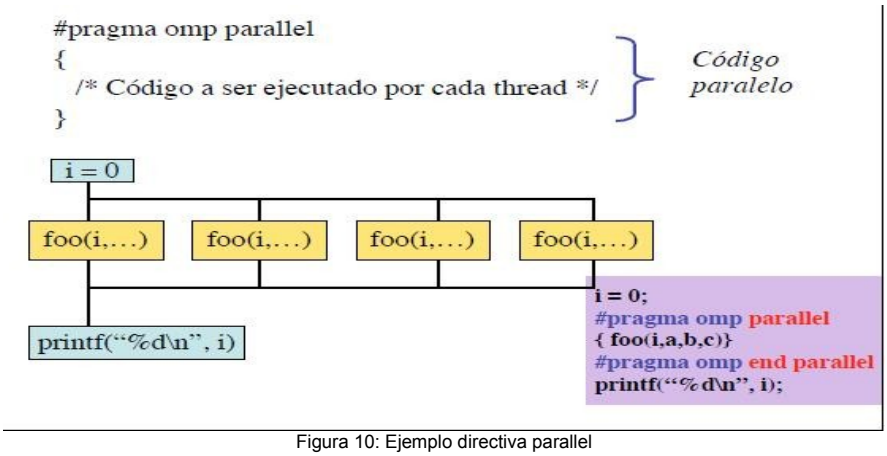


Figura 10: Ejemplo directiva parallel

- **for :** Indica que a cada thread se le asigna una parte del for a ejecutar (paralelismo de datos).
- **sections**: Indica que cada sección será ejecutada por un thread (paralelismo funcional).
- **single**: Indica que esta sección será ejecutada por un único thread.
- **master :** delimita un bloque estructurado que solo es ejecutado por el thread maestro. Los otros *threads* no lo ejecutan.
- **parallel for**: Combinación de parallel (fork) + for (repartición del for entre los threads).
- **parallel sections**: Combinación de parallel (fork) + sections (asignación de cada sección a un thread).
- **critical**: Indica que la sección será de exclusión mutua.
- **barrier**: Indica la necesidad de una sincronización de los threads en este punto.

Cláusulas a destacar

- **schedule** (static | dynamic | guided | runtime [, chunk]): Determina de que forma se realizará la asignación del trabajo a los threads. *(Figura 11)*

    - **static** : "chunk" iteraciones se asignan de manera estática a los *threads* en round-robin
    - **dynamic**: Cada *thread* toma "chunk" iteraciones cada vez que está sin trabajo
    - **guided** :Cada *thread* toma iteraciones dinámicamente y progresivamente va tomando menos iteraciones

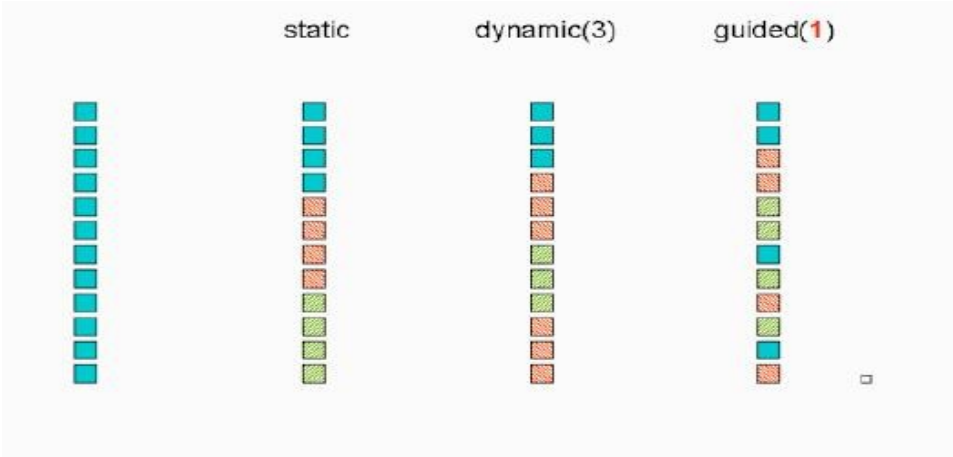


Figura 11: Asignación del trabajo dependiendo de la directiva schedule

- **private** (variable [,variable, …]): Indica que las variables que aparecen en la clausula seran privadas, cada thread tendrá una copia independiente.

En la figura 12 vemos un ejemplo de uso de OpenMP.

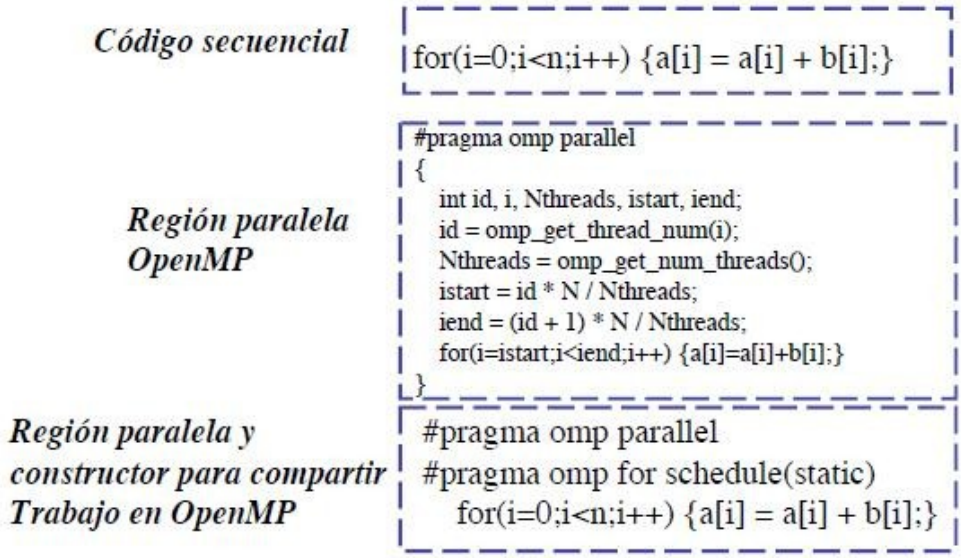


Figura 12: Ejemplo del uso de OpenMP en un programa escrito en el lenguaje de programación C

3. Algoritmo Needleman-Wunsch

En este capitulo se describirá el problema del alineamiento de secuencias para cadenas de ADN, utilizando como solución el algoritmo conocido como Needleman-Wunsch.

3.1. Marco del algoritmo dentro de la biología

Una de las claves fundamentales de los descubrimientos en la biología se basa en las comparaciones entre dos individuos diferentes para obtener conclusiones sobre ellas.

En la biología molecular se realiza el mismo tipo de comparaciones para obtener conclusiones, pero en este caso los individuos son secuencias de genes o de proteínas. En estos casos analizamos las similaridades y diferencias entre dos proteínas (o ADN) con el objetivo de deducir relaciones estructurales, funcionales o evolutivas entre ellas.

Una de las aplicaciones más importantes de comparar secuencias es la determinación de una similitud entre dos o más secuencias que justifique la inferencia de una homología evolutiva [6] entre ellas. La homología indica si dos secuencias son homólogas o no, dependiendo del grado de similitud y la significancia estadística del alineamiento. Es poco probable que dos proteínas con secuencias similares hayan evolucionado independientemente. Tales similitudes indican, por lo tanto, que las dos proteínas deben estar relacionadas y que comparten un ancestro común. Las proteínas relacionadas se dice que son homólogas.
La forma más común de comparar dos secuencias es realizar alineamientos de secuencias.

El alineamiento de secuencias normalmente se representan escribiendo las 2 secuencias que queremos comparar una encima de la otra *(Figura 13)*. Si en una posición nos encontramos la misma letra para ambas secuencias, esto significa que esa posición se ha conservado durante la evolución. En cambio, si las letras no coinciden o nos encontramos un hueco, esto se interpreta como una mutación puntual.

```
G-IISKILRKE-KGGYEITIVDASNERQVID
GKIVAITALSEKKGGFEVSIEKA-NGEVVVD
```

Figura 13 : Ejemplo de alineamiento entre 2 secuencias.

Existen dos formas de alinear dos secuencias: intentar encontrar los dos fragmentos de ambas secuencias que tienen un alineamiento con una puntuación máxima (*alineamiento local*) o intentar encontrar un alineamiento de las secuencias completas con una puntuación máxima (*alineamiento global*).
El alineamiento local es adecuado cuando las secuencias a comparar no se parecen a lo largo de toda su secuencia, mientras que el global se usa cuando las secuencias son muy parecidas entre ellas y de medidas parecidas.

En nuestro trabajo nos centraremos en uno de los algoritmos de alineamiento global más conocidos que existe, el algoritmo de Needleman- Wunsch [7]

3.2. Descripción del algoritmo

El algoritmo de Needleman-Wunsch fue propuesto por primera vez en 1970, por Saul Needleman y Christian Wunsch. Se trata de un ejemplo típico de programación dinámica, con la finalidad de obtener el alineamiento optimo de secuencias de proteínas o de ácidos nucleicos.

Sea el alfabeto ∑ = *{ A, G, C, T }* sobre el cual definimos 2 cadenas, *s1 = GAATTCAGTTA* y *s2 = GGATCGA*. El objetivo del algoritmo es alinear las 2 cadenas de forma que se pueda identificar en qué partes de las secuencias se produce una mutación o qué partes se mantienen indiferentes. El alineamiento sería el siguiente:

G	A	A	T	T	C	A	G	T	T	A
G	G	A	T	-	C	-	G	-	-	A

Según el alineamiento vemos que el segundo, quinto, séptimo, noveno y décimo carácter son diferentes para ambas secuencias, esto significa que han sufrido una mutación.
Para llevar a cabo esta tarea, el algoritmo se divide en 3 partes:

1) Inicialización de la matriz de puntuaciones.

2) Cálculo de la matriz de puntuaciones.

3) Realización del *traceback* y generación del alineamiento.

En la primera parte, se genera la matriz *M* en la cual se colocan todas las posibles combinaciones de las dos secuencias *s1* y *s2* más una fila y columna de ceros para la generación de la recursividad. Definimos *n* = |*s1*| y *m* = |*s2*| entonces la matriz *M* será de *(m + 1) x (n +1)*.
Por lo que en la posición *i* se encuentra el *i-1ésimo* nucleótido y en la posición *j* el *j-1ésimo* nucleótido *(Figura 14)*.

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	-	-	-	-	-	-	-	-	-	-	-
G	0	-	-	-	-	-	-	-	-	-	-	-
A	0	-	-	-	-	-	-	-	-	-	-	-
T	0	-	-	-	-	-	-	-	-	-	-	-
C	0	-	-	-	-	-	-	-	-	-	-	-
G	0	-	-	-	-	-	-	-	-	-	-	-
A	0	-	-	-	-	-	-	-	-	-	-	-

Figura 14 : Matriz de puntuaciones inicializada

El segundo paso, consiste en realizar el cálculo de cada elemento de la matriz de puntuaciones *(Figura 16),* aplicando la siguiente formula para cada una de las posiciones de la matriz :

$$M_{i,j} = \text{máximo} (M_{i-1, j-1} + S_{i,j} ; M_{i, j-1} + W ; M_{i-1, j} + W)$$

Donde :

- $M_{i-1, j-1} + S_{i,j}$: Indica la coincidencia o no coincidencia de los caracteres de la secuencia, si los caracteres coinciden $Si_{,j}$ sera un valor positivo, sino sera un valor negativo.
- $M_{i, j-1} + W$: Indica la suma en horizontal más la penalización por *gap (W)*.
- $M_{i-1, j} + W$: Indica la suma vertical más la penalización por *gap (W)*.

Es decir para calcular cada elemento de la matriz de puntuaciones necesitamos el valor de los elementos que se encuentran inmediatamente a la izquierda, arriba y en diagonal del elemento que queremos calcular. *(Figura 15)*

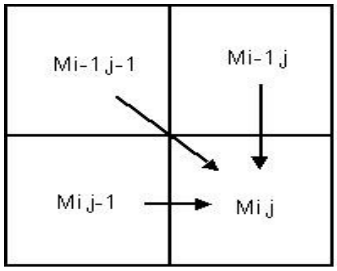


Figura 15: Dependencia de datos para el calculo del elemento Mi,j de la matriz de puntuaciones

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	4	3	3	3	4	5	5	5	5	3

Figura 16 :Cálculo de la matriz de puntuaciones

El tercer paso consiste en hacer el *traceback* *(Figura 17)*, el cual empieza desde el extremo inferior derecho de la matriz *M* y acaba en el extremo superior izquierdo. Se avanza siempre o hacía la diagonal izquierda o hacía arriba o hacía la izquierda, cogiendo siempre el valor mayor de las opciones precedentes. En caso de que el valor mayor sea el mismo en más de un vecino se le da prioridad a la diagonal izquierda.

		G	A	A	T	T	C	A	G	T	T	A
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	1	1	1	1	1	1	1	1	1	1	1
G	0	1	1	1	1	1	1	1	2	2	2	2
A	0	1	1	2	2	2	2	2	2	2	2	3
T	0	1	2	2	3	3	3	3	3	3	3	3
C	0	1	2	2	3	3	4	4	4	4	4	4
G	0	1	2	2	3	3	4	4	5	5	5	5
A	0	1	4	3	3	3	4	5	5	5	5	3

Figura 17: Traceback

Ya solo nos falta alinear las 2 secuencias a partir del *traceback*, el cual se construye de la siguiente manera :

Sea *a1* el alineamiento correspondiente a *s1* y *a2* el alineamiento correspondiente a *s2*. Comenzando con *a1= Φ* y *a2= Φ*

- Si el avance es diagonal, alineamos:
  $a1 = a1 + s1_{i-1}$
  $a2 = a2 + s1_{i-1}$

- Si el avance es horizontal, alineamos :
  a1= a1 + s1i-1
  a2 = a2 + -

- Si el avance es vertical, alineamos :
  a1 = a1 + -
  $a2 = a2 + s1_{i-1}$

4. Estudio y optimización del algoritmo Needleman-Wunsch serie

En este capítulo se analizará el algoritmo de alineamiento de secuencias Needleman-Wunsch, se presentarán una serie de optimizaciones y finalmente se presentarán los resultados en forma de tiempo de ejecución del programa.

El algoritmo que se tomará como punto de partida es un pseudocódigo [8], el cual ha sido adaptado al lenguaje C. El código resultante no es todo lo eficiente que quisiéramos, por lo tanto se le aplicarán una serie de optimizaciones con la finalidad de disminuir su tiempo de ejecución, reduciendo la cantidad de operaciones primitivas a realizar.

En este capítulo solo consideraremos la ejecución del programa sobre un único núcleo del procesador.

4.1. Análisis de la estructura del programa

La finalidad de este análisis es identificar la fase más significativa de nuestro programa.
El tiempo de ejecución depende de factores variados y, muy en particular, del tamaño de los datos de entrada o más bien del tamaño problema.
Por lo tanto el primer paso para analizar el algoritmo será identificar los parámetros que determinan el tamaño del problema.
Para el algoritmo Needleman-Wunsch el tamaño del problema (*Problem Size*) se define por :

- ***N:*** longitud de la secuencia 1
- ***M***: longitud de la secuencia 2

La estructura de nuestro programa se muestra en la siguiente figura *(Figura 18)*. Donde la función alinear es el programa que queremos analizar.

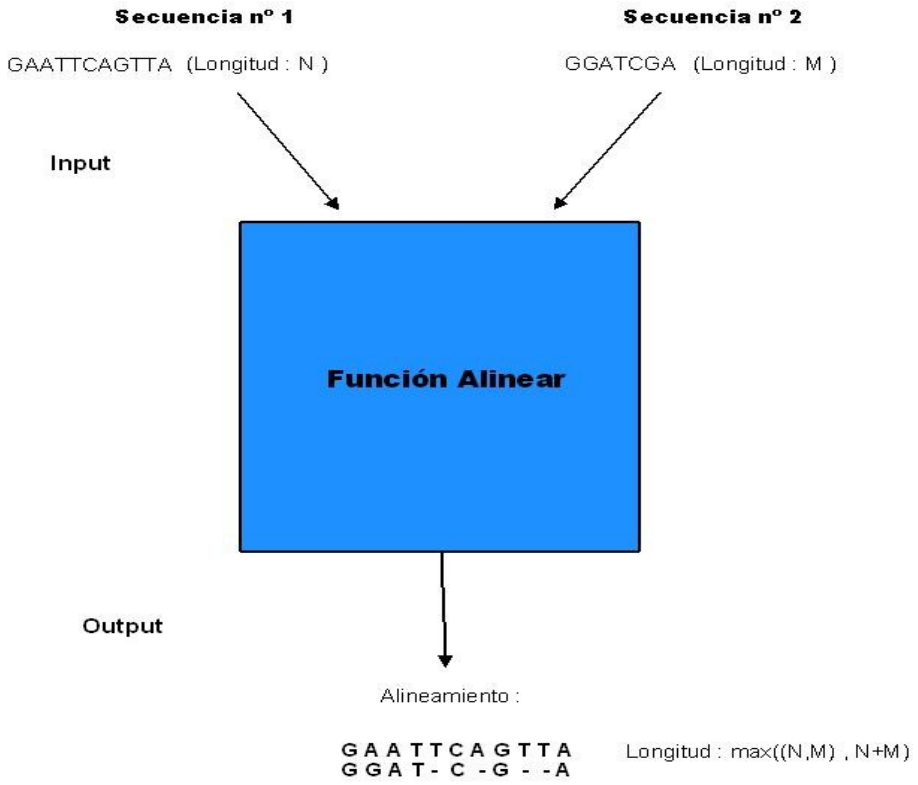


Figura 18 : Estructura del programa

Analizando los bucles del código identificamos 3 fases que comprenden la función alinear *(Figura 19)*, las cuales se ejecutan de forma secuencial.

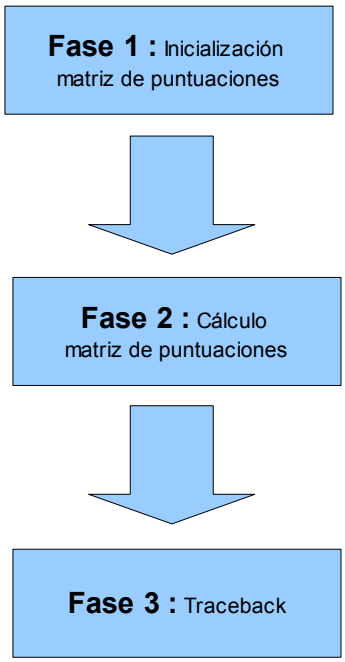


Figura 19 : Fases de la función alinear

Si procedemos a calcular la complejidad de cómputo de cada una de las fases en función del tamaño de las secuencias, obtenemos :

- Fase 1 : Esta es de ***O (N + M)*** operaciones de cómputo. Esto es debido a que tenemos que recorrer la matriz de puntuación inicializando a 0 la primera fila de tamaño N y la primera columna de tamaño M.
- Fase 2 : Esta es de ***O (N x M)*** operaciones de cómputo. Esta complejidad es debido a que hay que realizar el proceso de comparación entre cada pareja de elementos de cada una de las secuencias y en total tendremos N x M parejas.
- Fase 3 : Es de ***O (N + M)*** operaciones de cómputo ya que tan solo hemos de recorrer hacia atrás N + M elementos de la matriz para obtener el alineamiento de mayor similitud entre las 2 secuencias.

Los requerimientos de memoria crecen linealmente respecto a M y N. No dependen ni del número de fases, ni del número de iteraciones que realizamos. Concretamente se necesita una matriz de tamaño N x M y dos vectores de tamaño N y M. Por lo tanto la complejidad de memoria para las 3 fases sera de ***O(N x M)***.

Para finalizar el análisis inicial del programa se ha ejecutado y se ha medido el porcentaje de tiempo que tarda en ejecutarse cada fase. La ejecución se ha hecho con valores de N y M = 100 y N y M = 1.000. Estos valores se escogen porque, como se mostrará más adelante, representan casos significativos.

Tamaño de las secuencias	Fase 1	Fase 2	Fase 3
N = 100 , M = 100	18%	80%	2%
N = 1.000, M = 1.000	13%	86%	1%

Por lo tanto, podemos concluir que la Fase 2 es la parte más significativa de nuestro programa, ya que además de ser la que realiza el mayor número de operaciones, como se puede observar por su complejidad, es la que tarda más tiempo en ejecutarse.
Por lo tanto en los siguientes apartados nos centraremos específicamente en ella.

4.2. Fase crítica

Como hemos visto en el apartado anterior, la fase crítica de nuestro programa es la encargada del cálculo de la matriz de puntuaciones. En este apartado se analizará en detalle esta parte del programa.

4.2.1. Análisis del bucle crítico

A continuación se muestra el pseudo-código de la primera versión del bucle critico y las variables utilizadas para representar el problema. El algoritmo resultante no es todo lo eficiente que debería ser y posteriormente se le aplicarán una serie de optimizaciones.

Código Fase crítica :

```
for(i=1;i<M+1;i++){

      for(j=1;j<N+1;j++){

            match= MATCHING(s1[j-1],s2[i-1]);
            opcio1 = matriu[((i-1)*(N+1)) + j-1] + match;
            opcio2 = matriu [(i*(N+1)) + j-1] + gap;
            opcio3 = matriu [((i-1)*(N+1)) + j] + gap;
            matriu[(i*(N+1)) + j] = MAX(opcio1, MAX(opcio2,opcio3));

      }
}
```

Variables :

- M : longitud de la primera secuencia a alinear
- s1 : primera secuencia
- N : longitud de la segunda secuencia a alinear
- s2 : segunda secuencia.
- matriu[1..m][1..n] : Vector que contiene la matriz de puntuaciones.
- gap : valor de penalización por alinear la secuencia con un agujero.
- match : valor de coincidencia de secuencias.

- opcio1: variable donde se guarda el valor del vecino que se encuentra en la posición diagonal izquierda al valor que queremos calcular.
- opcio2: variable donde se guarda el valor del vecino que se encuentra en la posición superior al valor que queremos calcular.
- opcio3: variable donde se guarda el valor del vecino que se encuentra a la izquierda del valor que queremos calcular.

Funciones :

- int Matching (char a, char b) : Compara el primer parámetro con el segundo y nos retorna un valor positivo (2) en caso que sean iguales o un valor negativo (-1) en caso que no lo sean.

```
#define MATCHING(A,B) (((A) == (B)) ? (2) : (-1))
```

- int MAX (int a, int b) : Compara *a* con *b* y retorna el que sea mayor de los 2.

```
#define MAX(A, B) (((A) > (B)) ? (A) : (B))
```

Las funciones se han definido como una macro para que cuando el programa se compile el código generado para la función se inserte en el punto donde se invoca a la función, en lugar de hacerlo en otro lugar y tener que hacer una llamada.

Definirlo de esta manera nos permite que el código de estas funciones se ejecute más rápido, ya que evita usar la pila para pasar parámetros, a la vez que también evitamos las instrucciones de salto y retorno que se generan al definir funciones externas al código principal.

Las operaciones en el interior del bucle mostrado se ejecutan N x M veces. Queremos estimar el número de operaciones realizadas. Al contar el número de operaciones no se han tenido en cuenta las operaciones con enteros ya que estas son difíciles de estimar debido a que varían en función de cómo optimiza el código el compilador y de la arquitectura usada. La arquitectura usada influye ya que no podemos saber una operación de alto nivel exactamente a cuántas operaciones de bajo nivel corresponde.

Para estimar las operaciones de lectura y escritura en memoria consideraremos que las variables escalares locales (*match,opcio1,opcio2,opcio3*) se corresponderán en la implementación del programa con registros de propósito general del procesador. Por lo tanto, los accesos a estas variables supondremos que no requieren la ejecución de operaciones de acceso a memoria. Esto sólo será cierto si el número de registros del procesador es suficiente para contener todas las variables escalares que se están utilizando durante ciertos periodos de tiempo.
Por otro lado, se considera que los vectores (*matriu,s1,s2*) se almacenan en la memoria, y no en registros, y que por lo tanto todos los accesos a un vector suponen operaciones de acceso a memoria.
Supondremos 1 salto condicional para controlar el bucle interior.

La cantidad de operaciones que se realizan en el bucle principal son las siguientes :

LoadMem :	5
StoreMem :	1
Salto Cond :	1
Integer Op :	**

4.2.2. Análisis del patrón de accesos a memoria

Si observamos los accesos a memoria que hace el bucle principal, nos encontramos en que tenemos 5 operaciones de lectura (LOAD) y 1 de escritura (STORE) para calcular cada elemento de la matriz de puntuaciones. Cada una de las 6 operaciones se ejecuta N x M veces.

Concretamente los accesos a memoria que se realizan son 3 LOADS (*matriu[i-1][j-1]* , *matriu[i][j-1]* y *matriu[i-1][j]*) y 1 STORE (*matriu[i][j]*) para calcular cada elemento de la matriz de puntuaciones y 2 LOADS (*s1[j–1]* y *s2[i–1]*) para traernos a memoria las 2 secuencias que vamos a comparar entre ellas para el posterior alineamiento.

Para calcular cada elemento de la matriz de puntuaciones necesitaremos los valores que se encuentran a la izquierda, arriba y en diagonal del elemento que queremos calcular. En la figura 20 podemos ver el patrón de accesos a memoria. A la hora de recorrer la matriz accedemos a posiciones inmediatamente consecutivas por filas (stride = 1, mejor caso posible) lo cual aumenta la localidad espacial de acceso a los datos.

Como veremos más adelante, el patrón de acceso secuencial es el más favorable al rendimiento en los procesadores. En realidad, como se trata de un patrón de acceso muy común, los procesadores están optimizados para ser eficientes con este tipo de accesos.
Por contra, tenemos poca localidad temporal, ya que en general sólo accedemos 1 vez para escribir y 3 veces para leer cada elemento de la matriz de puntuaciones.

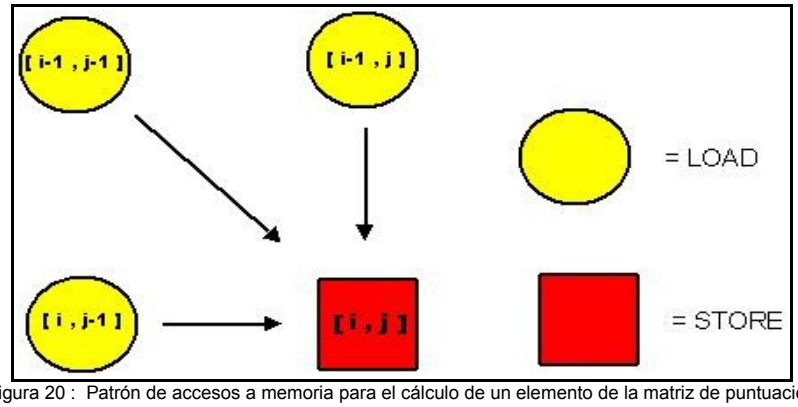


Figura 20 : Patrón de accesos a memoria para el cálculo de un elemento de la matriz de puntuación

A la hora de comparar los elementos de cada una de las secuencias entre sí, podemos observar *(Figura 21)* que el elemento 1 de la secuencia 1 se compara con los N elementos de la secuencia 2, y así con cada uno de los elementos que forman la secuencia 1.

Este patrón de accesos a memoria tiene una buena localidad temporal ya que constantemente estamos referenciando posiciones de memoria que ya han sido accedidas en un “pasado cercano”. En concreto para la secuencia 1 accederemos M veces consecutivas al mismo elemento y para la secuencia 2, accedemos N veces al mismo elemento, pero recorriendo todos los elementos antes de volver a acceder al mismo.

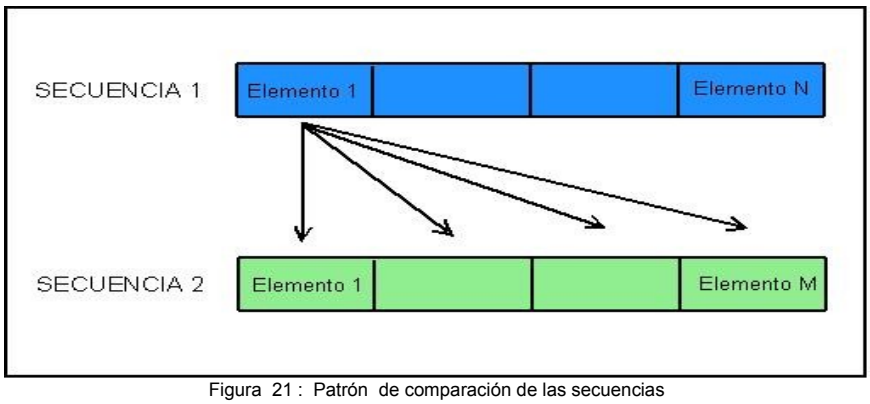


Figura 21 : Patrón de comparación de las secuencias

4.2.3. Análisis de saltos condicionales

Hay 1 salto condicional que se ejecuta N x M veces y su comportamiento es que siempre salta, excepto 1 vez de cada N ejecuciones. Un patrón de saltos en el que la mayor parte de las veces el resultado es el mismo, como es nuestro caso, es bueno ya que favorece el rendimiento de los procesadores que realizan predicción de saltos. Como en el caso de los accesos secuenciales a memoria, los procesadores están optimizados para ser eficientes con este tipo de patrones de comportamiento de saltos.

4.2.4 Análisis del tipo de datos usados

A la hora de trabajar se ha analizado la viabilidad de utilizar el tipo de datos *signed short* en favor del tipo de datos *signed int,* para la matriz de puntuación.

La ventaja de usar *signed shorts* es que ocupan la mitad que los *signed int*, 2 bytes frente a los 4 bytes de los enteros. Por lo tanto las variables ocupan menos memoria lo que provoca un ahorro de memoria. Los cálculos también se realizan más rápido.

Como inconveniente nos encontramos que los signed shorts solo pueden contener números de -32.768 a +32.767. Mientras que los signed int pueden representar del -2.147.483.648 a +2.147.483.647.

Si suponemos el caso de N = 8400 y M = 8400, el número más grande que podría contener la matriz de puntuaciones sería aproximadamente el 16.800
Por lo cual deducimos que si usamos *signed short* podremos alinear secuencias de tamaño 16.383 caracteres.

Hemos hecho pruebas iniciales y la ganancia en forma de tiempo de ejecución es muy poca. A partir de aquí consideraremos que los datos de la matriz de puntuaciones son siempre *signed int*

4.3. Optimizaciones

Se han implementado una serie de optimizaciones sencillas sobre el algoritmo, con la finalidad de reducir el número de instrucciones ejecutadas y mejorar el tiempo de ejecución. En este apartado explicaremos como funcionan, cual es su objetivo y como han sido implementadas dentro de nuestro programa.

4.3.1. Code Motion

El Code Motion consiste en mover instrucciones invariantes fuera del cuerpo del bucle sin afectar la semántica del programa. El código movido fuera del cuerpo del bucle (cálculo o acceso a memoria) se ejecuta con menos frecuencia, reduciendo el tiempo total de ejecución.

El objetivo de esta optimización es intentar reducir el número de instrucciones que se ejecutan dentro del bucle crítico.

En nuestro caso lo hemos implementado moviendo las siguientes instrucciones fuera del bucle :

a. *s2char = s2[i-1];* La posición de la secuencia en la que nos encontramos se lee fuera del bucle y se guarda en una variable local.

b. *c= (i-1)*(N+1); c2 = i*(N+1);* Calculamos estas expresiones fuera del bucle.

Es posible que alguno de estos movimientos de instrucciones el propio compilador ya los saque fuera del bucle, pero como podremos ver en el capítulo dedicado a los resultados los cambios que hemos realizado son significativos ya que reducen el tiempo de ejecución del código. La razón es que el compilador no sabe si los vectores *matriu*, *s1* y *s2* se solapan en memoria.

Código fase crítica :

```
for(i=1;i<M+1;i++){

        c= (i-1)*(N+1);
        c2 = i*(N+1);
        s2char = s2[i-1];

        for(j=1;j<N+1;j++){

                match= MATCHING(s1[j-1],s2char);
                opcio1 = matriu[c + j-1] + match;
                opcio2 = matriu [c2 + j-1] + gap;
                opcio3 = matriu [ c + j] + gap;
                matriu[c2 + j] = MAX(opcio1, MAX(opcio2,opcio3));

        }
}
```

4.3.2. Loop Unroll

La técnica de Loop Unrolling consiste en agrandar el cuerpo de los ciclos para hacer una mayor cantidad de cosas en cada paso de iteración. La efectividad de la estrategia se basa en el pipelining del procesador de la siguiente manera: un ciclo que itera mil veces sobre una instrucción nunca deja que se llene al pipeline (cada vez que se vuelve a ejecutar el cuerpo del

ciclo el pipeline se vacía), pero si el ciclo itera solo 10 veces y realizando 100 operaciones en cada iteración, entonces el pipeline puede aprovecharse en cada iteración, resultando en una clara mejora de eficiencia.

El objetivo de esta optimización es reducir el número de instrucciones de acceso a memoria que hace el programa.

- Load Data Reuse dentro del bucle

  Como vemos en la figura 22 al realizar 2 iteraciones dentro del mismo bucle podemos reaprovechar los valores R3 y RES1 para calcular RES2, lo que nos da un ahorro de 2 accesos a memoria cada 2 elementos calculados en relación con la versión sin loop unrolling.

  Por tanto, de media en cada iteración tendremos 4 LOAD en lugar de los 5 que teníamos en el bucle inicial (reducción del 20% en número de LOAD)

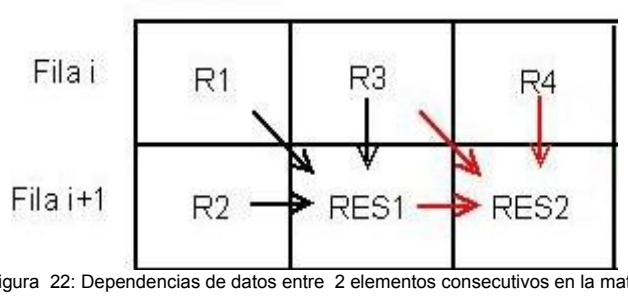


Figura 22: Dependencias de datos entre 2 elementos consecutivos en la matriz de puntuaciones

Código fase critica:

```
            for(j=1;j<N+1;j=j+2){

                  match1 = MATCHING(s1[j-1],s2char);
                  match2 = MATCHING(s1[j],s2char);

                  r1 = matriu[c+ j-1];
                  r2 = matriu[c2 + j-1];
                  r3 = matriu[c+ j];

                  opcio1 = r1 + match1;
                  opcio2 = r2 + gap;
                  opcio3 = r3 + gap;
                  resultat1 = MAX(opcio1, MAX(opcio2,opcio3));

                  r4 = matriu[c + j+1];
                  opcio1 = r3 + match2;
                  opcio2 = resultat1 + gap;
                  opcio3 = r4 + gap;

                  matriu[c2 + j+1]  = MAX (opcio1, MAX(opcio2,opcio3));
                  matriu[c2 + j] = resultat1;

            }
      }
```

- Load Data Reuse entre iteraciones

Podemos aplicar la misma técnica de reaprovechar valores, que hemos aplicado en el apartado anterior, pero en vez de dentro del bucle entre iteraciones. Es decir, si nuestros datos se representan de la siguiente manera *(Figura 23) :*

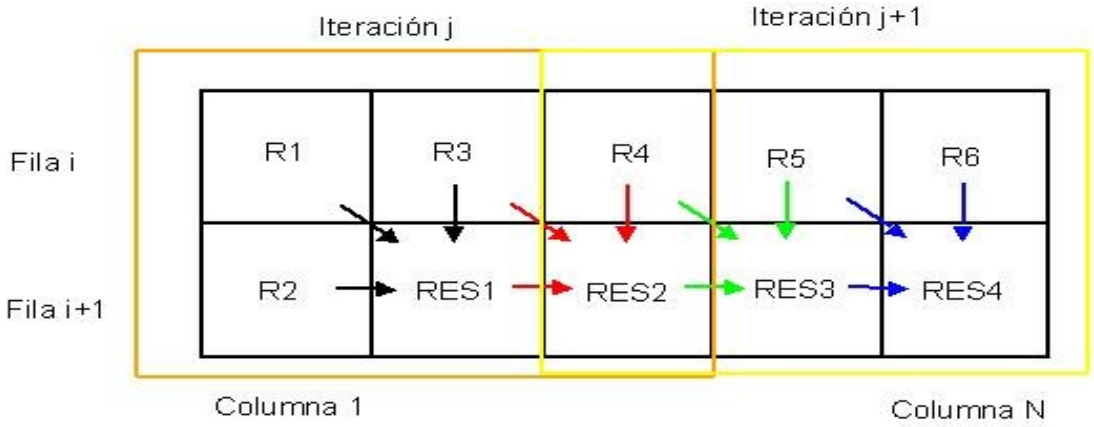


Figura 23: Dependencias de datos entre 2 iteraciones del bucle

En la iteración j, calculamos RES1 y RES2, para ello necesitamos los valores R1, R2, R3 y R4, en la iteración j+1 calcularemos RES3 y RES4 para ello necesitamos R4 , R5, R6 y RES2. En este punto nos damos cuenta que en la iteración j+1 accedemos a memoria para ir a buscar datos que ya teníamos en la iteración j (R4 y RES2).
Por lo tanto si en la iteración j guardamos los valores R4 y RES2 en 2 registros, los podremos usar en la siguiente iteración sin necesidad de acceder a memoria para traérnoslos.

A nivel de instrucciones este hecho nos ahorrará 2 lecturas a memoria. Y este paso lo podemos aplicar a todas las iteraciones del programa excepto a los 2 primeros elementos de cada fila.

Por tanto, de media en cada iteración tendremos 3 LOAD en lugar de los 5 que teníamos en el bucle inicial (reducción del 40% en número de LOAD)

Código fase critica:

```
            for(j=1;j<N+1;j=j+2){

                  r3 = matriu[c+ j];
                  match1 = MATCHING(s1[j-1],s2char);
                  match2 = MATCHING(s1[j],s2char);

                  opcio1 = r1 + match1;
                  opcio2 = r2 + gap;
                  opcio3 = r3 + gap;
                  resultat1 = MAX2(opcio1, MAX2(opcio2,opcio3));

                  r1 = matriu[c + j+1];
                  opcio1 = r3 + match2;
                  opcio2 = resultat1 + gap;
                  opcio3 = r1 + gap;
                  r2 = MAX (opcio1, MAX2(opcio2,opcio3));

                  matriu[c2 + j] = resultat1;
                  matriu[c2 + j+1] = r2;
            }
      }
```

4.4. Experimentación

En este apartado se describirán los procesadores utilizados en este trabajo. A continuación se presentará el método experimental para tomar medidas de rendimiento de cada procesador y los workloads utilizados.

4.4.1. Sistemas de cómputo

Para cada procesador se muestra el nombre clave, su arquitectura, el modelo de ejecución de instrucciones, los tamaños de cada nivel de memoria caché, el sistema operativo y el compilador usado.

Intel Pentium D

Procesador	Intel Core 2 Duo @ 3.00 GHz
Arquitectura	32 bits
Ejecución de instrucciones	Fuera de orden
Cache L1 de instrucciones	32 Kbytes
Cache L1 de datos	32 Kbytes
Cache L2	2048 Kbytes
Compilador :	GCC 4.33 y ICC 11.1
Sistema Operativo :	Ubuntu
Kernel	2.6.28-15

Intel Pentium Core 2 Duo

Procesador	Intel T8100 Duo @ 2,1 GHz
Arquitectura	64 bits
Ejecución de instrucciones	Fuera de orden
Cache L1 de instrucciones	32 Kbytes
Cache L1 de datos	32 Kbytes
Cache L2	3072 Kbytes
Compilador	GCC 4.3.2
Sistema Operativo :	Debian
Kernel	2.22.3

Sun UltraSPARC T2

Procesador	SUN SPARC-E-T5120, UltraSPARC T2 @ 1165 MHz
Arquitectura	64 bits
Ejecución de instrucciones	En orden
Cache L1 de instrucciones	16 Kbytes
Cache L1 de datos	8 Kbytes
Cache L2	4096 Kbytes
Compilador	Sun C 5.9 SunOS Sparc
Sistema Operativo :	Sun OS
Kernel	02/06/24

4.4.2. Métodos de medición y análisis

Tanto los sistemas operativos como los lenguajes tienen instrucciones que permiten medir el tiempo total de ejecución de un programa.
Pero nuestro interés recae solamente en medir la parte del programa que realiza el mayor número de operaciones, es decir, la parte de mayor complejidad con la mayor precisión posible.
Por este motivo se han añadido al algoritmo funciones concretas que miden solamente esta parte del código.

Para medir tiempos en el procesador Sun UltraSPARC hemos utilizado la función *gettimeofday*

- *int gettimeofday(struct timeval *tv, struct timezone *tz) : Esta función nos permite obtener la hora del sistema operativo en el momento de su llamada. La precisión con la que trabaja es de microsegundos.*

A la hora de trabajar con ella la usamos antes y después de la ejecución en la zona que queremos medir el tiempo. Luego tan solo hay que restar los valores entre sí para obtener el tiempo que ha tardado esa zona en ejecutarse.

Para medir tiempos en procesadores x86 hemos utilizado una instrucción maquina especial.
Estas instrucciones hacen uso del acumulador principal EAX y del secundario EDX de la arquitectura Intel x86 para obtener el número de ciclos totales.

A la hora de trabajar con ellos, se llama a la función que inicializa los acumuladores a 0 antes de la ejecución de la fase que queremos medir, y se llama a la función que nos calcula el número total de ciclos acumulados al final de la ejecución de la fase a medir.

4.4.3. Métricas y Workloads

La métrica fundamental que se ha medido en las ejecuciones ha sido el tiempo de ejecución del programa.

En las gráficas el eje Y muestra el tiempo de ejecución dividido por la complejidad, en nanosegundos mientras que el eje X están expuestas las versiones del programa. Dividimos el tiempo entre la complejidad ya que así se anula el efecto de crecimiento en el tiempo de ejecución debido al incremento del volumen de operaciones.

El programa usado para medir los tiempos es la versión con todas las optimizaciones discutidas en el apartado anterior.

Las ejecuciones se realizaran con tamaños de secuencia que van de los 50 hasta los 10.000 caracteres de longitud. Se han hecho las pruebas alineando más de una secuencia y con tamaños diferentes. En todos los casos el número total de iteraciones que tiene que hacer el programa es el mismo, pero como se podrá apreciar los resultados varían dependiendo del tamaño de las secuencias. El número de tamaños y secuencias usados son :

- Alineamos 2 secuencias de longitud 10.000 entre ellas.
- Alineamos 100 secuencias de longitud 200 entre ellas.
- Alineamos 400 secuencias de longitud 50 entre ellas.

Estos workload se escogen porque, como se verá mas adelante, representan casos significativos.

4.5. Resultados

En este apartado se mostrarán y discutirán los resultados de rendimiento obtenidos al ejecutar la implementación del algoritmo Needleman-Wunsch en serie en los tres procesadores descritos anteriormente.

4.5.1. Evolución de tiempos de las diferentes versiones del programa en la arquitectura x86-32

Se han realizado pruebas sobre el procesador Intel con arquitectura de 32 bits usando diferentes versiones del programa para comprobar que realmente las optimizaciones propuestas en el apartado anterior aportan una mejora a nivel de tiempo de ejecución. Las versiones utilizadas son:

- La versión “normal” es la versión que contiene tan sólo la mejora del code-motion.
- La versión “Loop Unroll” contiene las mejoras de code-motion y el Loop Unroll con Load Data Reuse dentro del bucle
- La versión “Loop Unroll mejorado” es la versión mas optimizada del algoritmo en serie. Ya que contiene las mejoras de code-motion y el Loop Unroll con Load Data Reuse entre iteraciones.

La figura 24 nos muestra el tiempo de ejecución de las diferentes versiones del programa usando el compilador GCC.

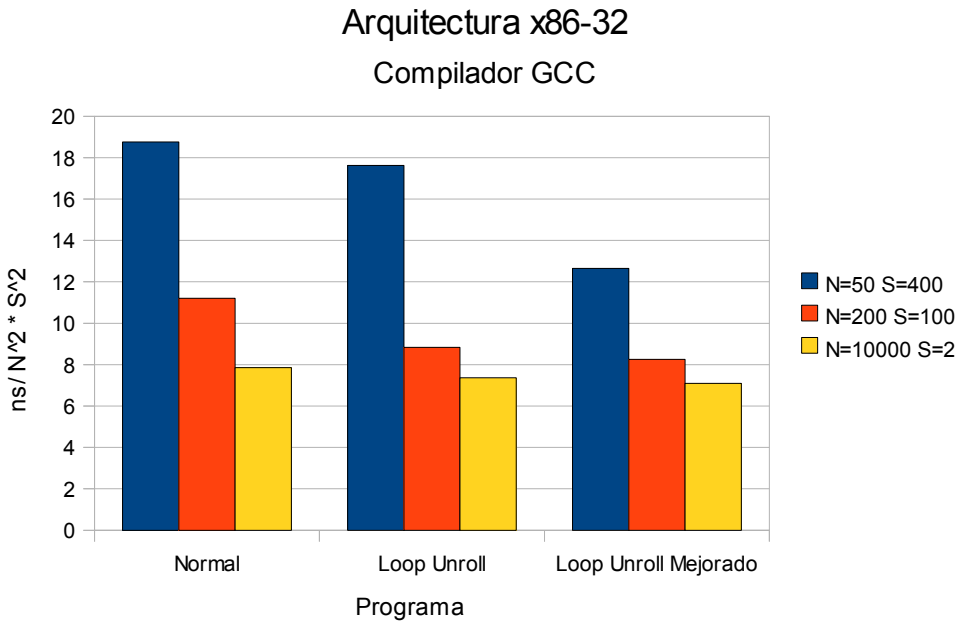


Figura 24: Tiempo de ejecución divididos por complejidad para la ejecución del algoritmo serie en el procesador x86-32 utilizando el compilador GCC con diferentes versiones del programa.

Podemos apreciar como las optimizaciones planteadas en el apartado anterior, mejoran el tiempo de ejecución, siendo la versión que contiene todas las optimizaciones planteadas (Loop Unroll mejorado) la que funciona sustancialmente mejor sobretodo en el peor caso (N=50 y S = 400).

Hemos realizado las mismas mediciones utilizando otro compilador como es el *Intel C Compiler*, y los resultados los podemos apreciar en la figura 25.

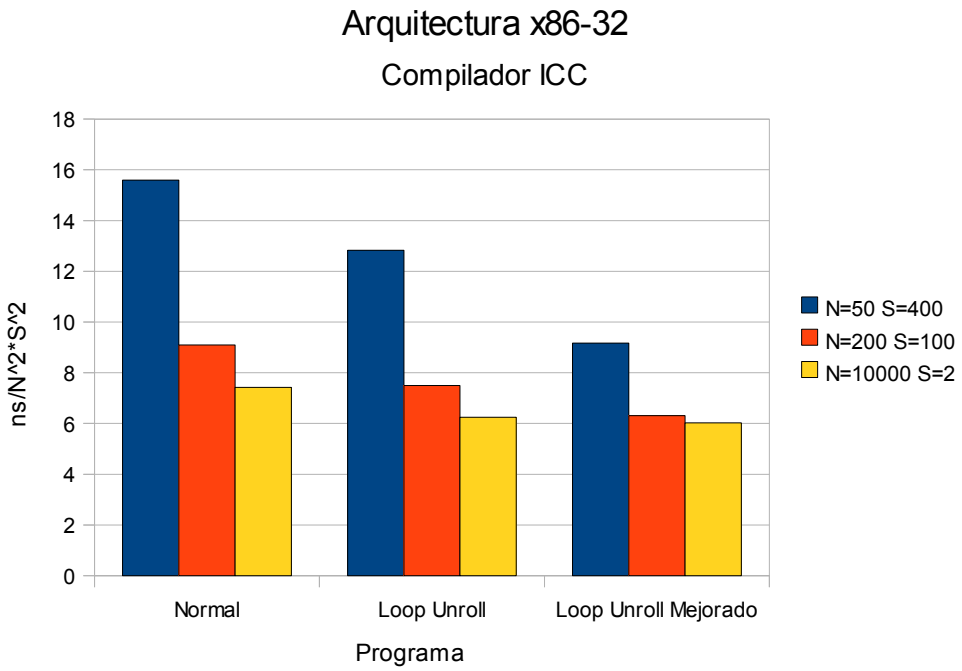


Figura 25: Tiempo de ejecución divididos por complejidad para la ejecución del algoritmo serie en el procesador x86-32 utilizando el compilador ICC con diferentes versiones del programa.

Si comparamos tiempos entre las 2 versiones vemos como la versión compilada con ICC es ligeramente más rápida, esto es debido a que este compilador genera un ensamblador más eficiente que el GCC. Ya que si miramos el número de instrucciones de la mejor versión (Loop Unroll mejorado) tenemos :

- GCC: Icount : 55 instrucciones x N x M Instrucciones
- ICC : Icount 47 instrucciones x N x M instrucciones

Una de las grandes diferencias es que el compilador ICC es capaz de utilizar las instrucciónes CMOV (Conditional Move) por contra del GCC que necesita de 3 instrucciones para realizar la misma tarea que hace el CMOV, de las cuales una es un salto.

El compilador ICC trabajando con el procesador con arquitectura de 32 bits igualmente tiene un problema y es que no es capaz de sustituir todas las instrucciones de comparación + salto por un CMOV, tan sólo puede con las más básicas. Cosa que por el contrario los compiladores que trabajan con arquitecturas de 64 bits si que son capaces de realizar.

4.5.2. Resultados en las 3 arquitecturas

En la figura 26 se muestran los resultados obtenidos de ejecutar la versión más optimizada del programa sobre los 3 procesadores. El objetivo es comparar el rendimiento en los 3 procesadores.

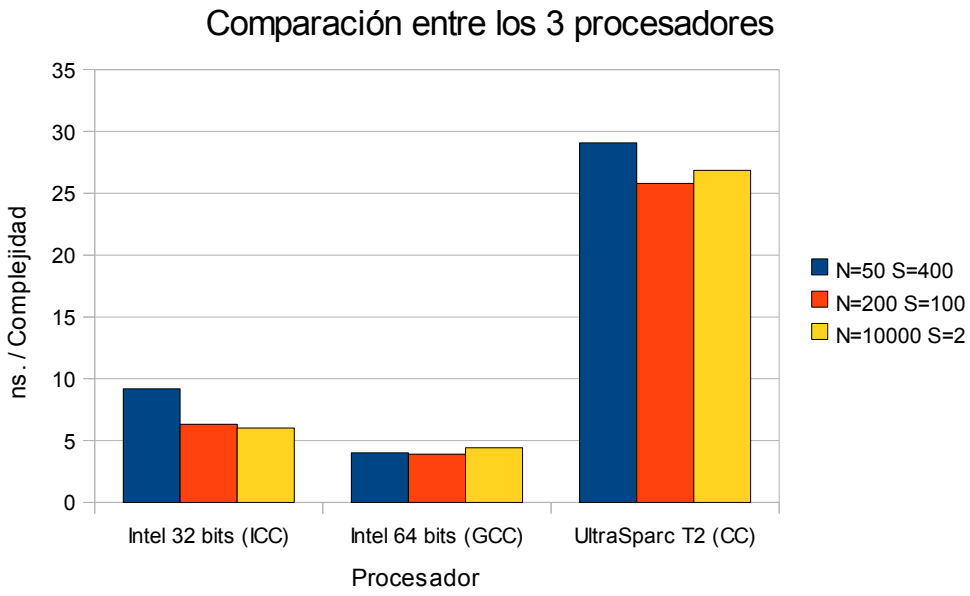


Figura 26: Tiempo de ejecución divididos por complejidad, para la ejecución del algoritmo serie más optimizado sobre los 3 procesadores

Para el procesador Intel de 32 bits lo que podemos apreciar es que sobre todo para el primer caso (N=50, S=400) el tiempo es mas elevado. Esto seguramente es debido al overhead causado por los bucles ya que esta versión al tener que alinear más secuencias ejecuta 6.000 veces más el bucle principal que la versión con N = 200 S = 100 y aproximadamente16.000 veces más que la versión con N=10.000 y S = 2.
Otro de los grandes problemas que tenemos al ejecutar el programa con esta arquitectura es el hecho de que tan solo tenemos 8 registros de propósito general con lo cual tenemos que estar haciendo continuamente movimientos de memoria a registros o entre registros porque nos quedamos sin registros para poder trabajar, lo cual hace aumentar el tiempo de ejecución, sobretodo si lo comparamos con la arquitectura de 64 bits.

Como podemos observar en el procesador Intel de 64 bits los tiempos se han reducido alrededor de un 60% para el primer caso y un 50% para los 2 últimos casos en comparación con el procesador con arquitectura de 32 bits, esto es debido a que el número de registros de propósito general se ha incrementado de 8 en los procesadores x86-32 a 16, y el tamaño de todos estos registros se ha incrementado de 32 bits a 64 bits.
Adicionalmente, el compilador trabaja de forma mucho más eficiente que su homólogo de 32 bits lo cual también influye en la mejora del tiempo de ejecución obtenido. Se ha analizado el código ensamblador generado por nuestro programa y los resultados son :

- Procesador Intel 32 bits : Icount : 47 instrucciones en el bucle interno
- Procesador Intel 64 bits : Icount: 35 instrucciones en el bucle interno

Por ultimo, a partir de la gráfica podremos ver como claramente el rendimiento que obtenemos en la maquina UltraSparc T2 es 7 veces inferior a los obtenidos en las arquitecturas Intel de 64 bits y unas 5 veces inferior al obtenido en la Intel de 32 bits.

Una de los grandes problemas que hace que empeore el rendimiento es el hecho de que esta arquitectura ejecuta las instrucciones en orden, al contrario que sus homólogas que son capaces de ejecutar fuera de orden. También hay que tener en cuenta que la UltraSparc tiene una frecuencia de reloj 2 veces inferior a la maquina de 64 bits y 3 veces inferior a la maquina de 32 bits con la que se han realizado las pruebas de medición.
Si observamos el ensamblador generado nos encontramos que éste está menos optimizado que el generado por el compilador GCC de 64 bits, y resulta bastante parecido al generado por el compilador ICC de 32 bits. Si contamos las instrucciones ensamblador generadas tenemos :

UltraSPARC T2 : Icount : 46 instrucciones x N x M

Procesador Intel 64 bits : Icount: : 35 instrucciones x N x M

Procesador Intel 32 bits : Icount:: 47 instrucciones x N x M

Las 3 máquinas son multicore pero en el algoritmo serie tan solo se trabaja en uno de sus cores, por lo que además no estamos utilizando las posibilidades que tiene la UltraSPARC de ejecutar hasta 8 threads por core.

4.5.3. Análisis de los factores que determinan el rendimiento

Uno de los indicadores para determinar el rendimiento y tiempo de ejecución de una aplicación es el CPI (Ciclos por Instrucción ejecutada). En nuestro caso hemos calculado el CPI de la mejor versión (Loop Unroll mejorado) y lo hemos comparado con el número de ciclos mínimo que requiere cada instrucción (CPI mínimo) en cada unos de los 3 procesadores. Los resultados son los siguientes :

Procesador x86-32

Núm. secuencias / longitud	Ciclos Totales	Tiempo (segs.)	Instrucciones del bucle	Ciclos por Resultado	CPI	CPI/CPI mínimo
S = 2, N = 10.000	7226545261	2,41	47	18,07	0,77	0,77 / 0,33
S = 100, N =200	7572779310	3,01	47	18,93	0,81	0,81 / 0,33
S = 400, N = 50	11015968804	3,67	47	27,54	1,17	1,17 / 0,33

Procesador x86-64

Núm. secuencias / longitud	Ciclos Totales	Tiempo (segs.)	Instrucciones del bucle	Ciclos por Resultado	CPI	CPI / CPI mínimo
S = 2, N = 10.000	3546187964	1,77	35	8,87	0,51	0,51 / 0,33
S = 100, N =200	3110415868	1,56	35	7,78	0,44	0,44 / 0,33
S = 400, N = 50	3209548549	1,6	35	8,02	0,46	0,46 / 0,33

Procesador UltraSPARC T2

Núm. secuencias / longitud	Ciclos Totales	Tiempo (microsegs)	Instrucciones del bucle	Ciclos por Resultado	CPI	CPI/CPI mínimo
S = 2, N = 10.000	16033476000	10739539	46	32,22	1,4	1,40 / 1
S = 100, N =200	15645067200	10317640	46	30,95	1,35	1.35 / 1
S = 400, N = 50	16123306800	11636166	46	34,91	1,52	1,52 / 1

El análisis del CPI permite hacer comparaciones sin considerar la frecuencia de reloj. En cada uno de los casos obtenemos :

- Intel 64 bits : 0,33 / 0,44 = 75 % del rendimiento pico del CPI

- Intel 32 bits : 0,33 / 0,77 = 42 % del rendimiento pico del CPI

  47 / 35 = 34% más instrucciones ejecutadas que la versión x86-64 bits.

- UltraSPARC T2 : 1 / 1,35 = 70% del rendimiento pico del CPI

  46 / 35 = 31% más instrucciones ejecutadas que la versión x86- 64 bits.

En la figura 27 podemos apreciar la comparación del CPI entre los 3 procesadores

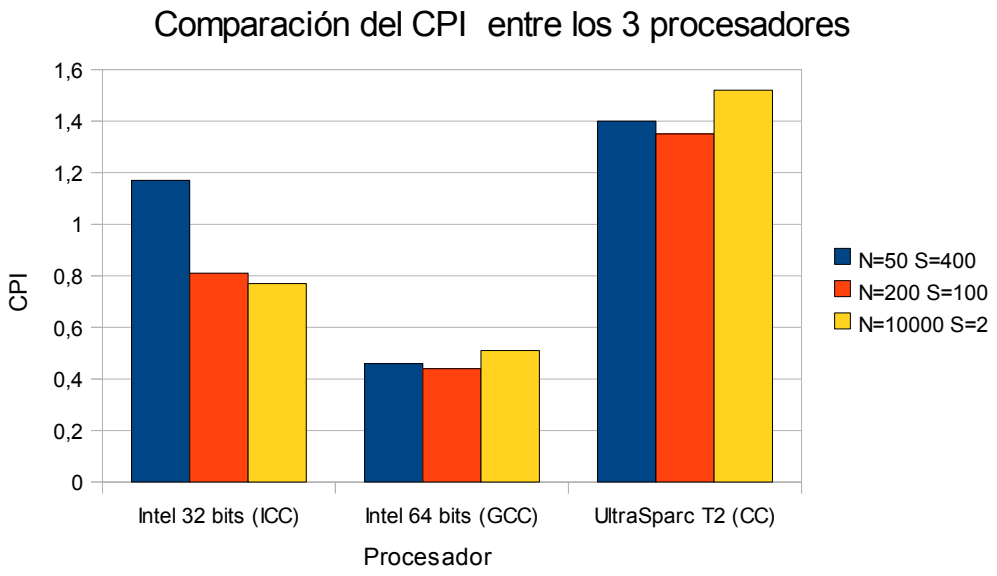


Figura 27: Comparación del CPI entre los 3 procesadores.

Los mejores resultados los obtenemos con el procesador Intel de 64 bits, viendo su CPI apreciamos que no hay problemas de dependencias de datos ni de latencias de ejecución que provoquen esperas. Podemos reducir el camino crítico pero no sirve de nada en este caso, ya que posiblemente el problema es que se está saturando algún recurso de cómputo, es decir el procesador es capaz de ejecutar 3 instrucciones por ciclo pero es probable que para algún tipo de instrucción tenga un límite más estricto.

En los otros procesadores hay margen de mejora, ya que como veremos en el siguiente apartado la UltraSPARC reduce mucho su tiempo de ejecución gracias al multithreading ya que consigue una eficiencia por thread de casi el 100%. En la arquitectura Intel 32 bits muchas de las anomalías de rendimiento se deben a la interacción con la jerarquía de memoria. En muchas ocasiones es debido a la confluencia de ciertos alineamientos en los accesos a memoria. Una forma de verificar que efectivamente el problema es causado por la jerarquía de memoria sería analizar los fallos de caché.

5. Diseño y estudio del algoritmo Needleman-Wunsch paralelo

En este capítulo se diseñará y analizará la paralelización del algoritmo de alineamiento de secuencias Needleman-Wunsch.

Se estudiarán 2 variantes que difieren entre la cantidad de cómputo y la de comunicación (granularidad). En la primera variante la comunicación entre los procesadores es poco frecuente y se realiza después de largos periodos de ejecución (granularidad gruesa). La segunda variante las tareas individuales son relativamente pequeñas en término de tiempo de ejecución y la comunicación entre los procesadores es frecuente (granularidad fina).

Finalmente se presentarán y analizarán los resultados de ambas variantes en forma de tiempo de ejecución del programa.

5.1. Paralelización large-grained

La paralelización de grano grueso se ha implementado basándonos en el hecho de que cuando alineamos secuencias genéticas normalmente se alinean dos o más secuencias entre ellas. Por lo tanto podemos repartir entre los threads el número de secuencias a alinear y que cada uno trabaje alineando una serie de secuencias diferentes.

Para repartir las secuencias entre los threads se ha implementado una función en C ajena a la función de alineamiento de secuencias. A continuación se muestra el pseudocódigo *simplificado* de la función

Pseudocódigo

```
#pragma omp parallel for default (shared) private(i)

for (i=0; i < (numero_de_secuencias * numero_de_secuencias ); i++)
{

      int s1 = i / numero_de_secuencias;
      int s2 = i % numero_de_secuencias;

      alinear(seq1(s1),seq2(s1));

}
```

En esta variante del algoritmo la asignación del volumen de cómputo a cada thread es equitativa. A partir de ahora llamaremos *nthreads* al número total de threads.

El bucle que se reparte entre los threads está compuesto de :

*Número_de_secuencias * Número_de_secuencias* elementos.

Al repartir el bucle, a cada thread le corresponden :

(*Número_de_secuencias * Número_de_secuencias) / nthreads*.

En la figura 28 se muestra como repartimos el trabajo si tenemos 3 secuencias a alinear y 2 threads.

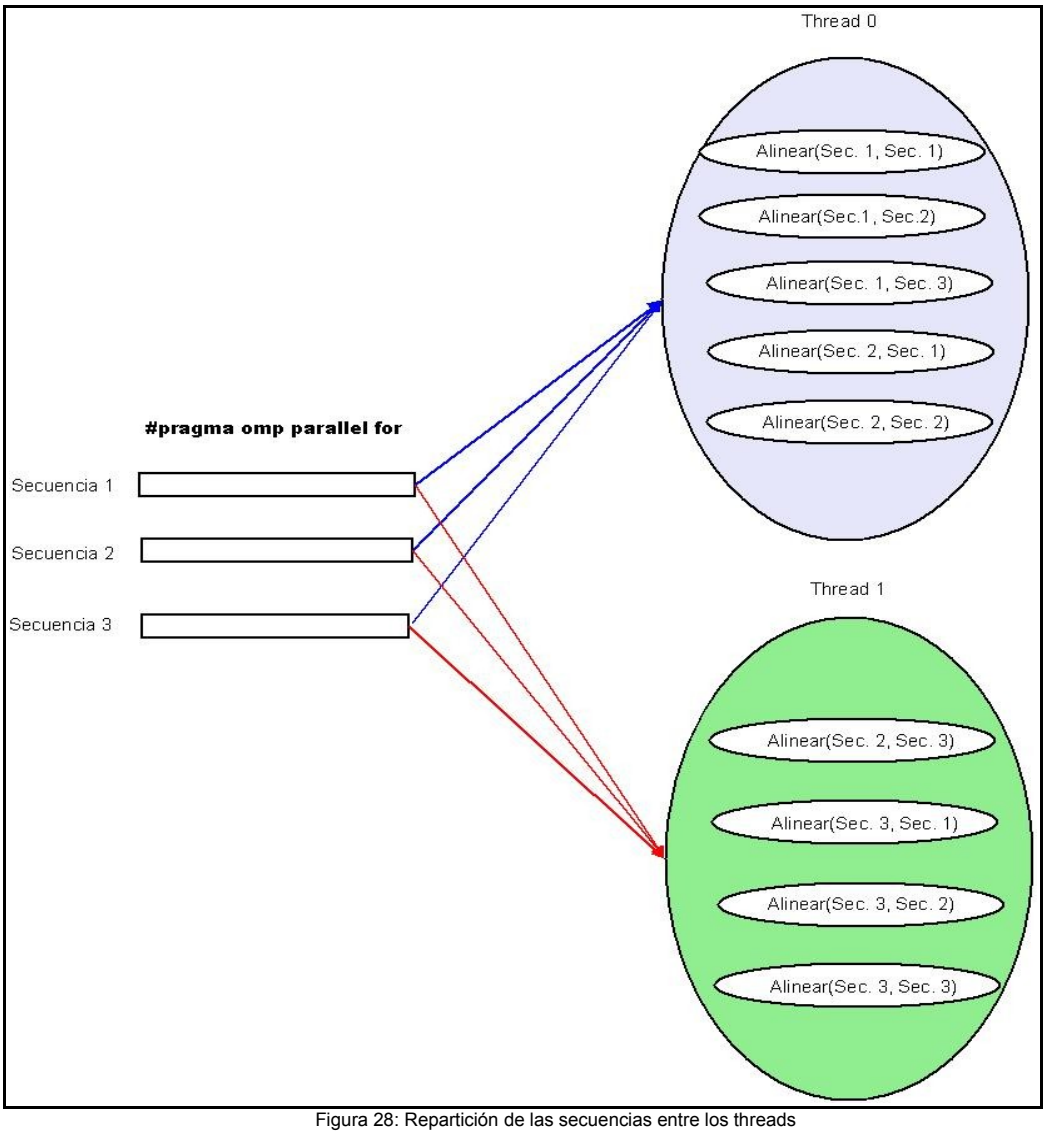


Figura 28: Repartición de las secuencias entre los threads

Sincronización

En esta variante del algoritmo paralelo los threads no deben esperar datos que provengan de otros threads por lo tanto no hay sincronización.

Requerimientos de memoria

Los requerimientos de memoria globales aumentan con el número de threads ya que cada thread escribe en una matriz de puntuaciones diferente, que es local a cada thread.
Los requerimientos de memoria para la función alinear crecen linealmente respecto al tamaño de las secuencias. Concretamente se necesita un vector de tamaño M (longitud de la secuencia 1) y un vector de tamaño N (longitud de la secuencia 2).

$$Memoria = N \times M \times nthreads + (N + M) \times nthreads$$

Comunicación

Cuando un thread de una región paralela escribe en una variable y posteriormente cualquiera de los otros threads de la región paralela necesita leer dicha variable, el sistema debe asegurar que el valor que se lee es el correcto. Las variables lógicas del programa se asignan a posiciones de memoria, que en un determinado momento del tiempo pueden corresponder con el contenido de una caché privada de un procesador, o de una posición de la memoria principal compartida. Por lo tanto, para que un thread lea el resultado producido por otro thread, puede ser necesario mover los datos entre los procesadores y las memorias caché. A este movimiento de datos entre threads le denominamos comunicación entre threads, y es importante determinar su volumen.

En cuanto a las comunicaciones, un mismo thread tiene dos comportamientos. Los threads generan datos (escribiendo resultados en memoria) y consumen datos (leyendo datos de la memoria, que previamente han sido escritos por otros threads).

En esta variante del algoritmo paralelo no tenemos comunicación entre los threads ya que los threads no escriben datos que vayan a ser leídos por otros threads. Esto es debido a que cada thread lee secuencias (que pueden ser las mismas) pero escribe en matrices de puntuación diferentes, que son locales a cada thread, y que sólo lee el thread que escribe.

5.2. Paralelización fine-grained

La paralelización de grano fino tiene sentido cuando sólo es necesario alinear una pareja de secuencias, además veremos que para que sea efectivo las secuencias deben ser largas.
Esta variante se ha implementado paralelizando la fase donde calculamos la matriz de puntuaciones, que como vimos en el capitulo anterior es la fase crítica del programa.
A la hora de paralelizar esta fase nos encontramos con las siguientes dependencias de datos *(Figura 29)*:

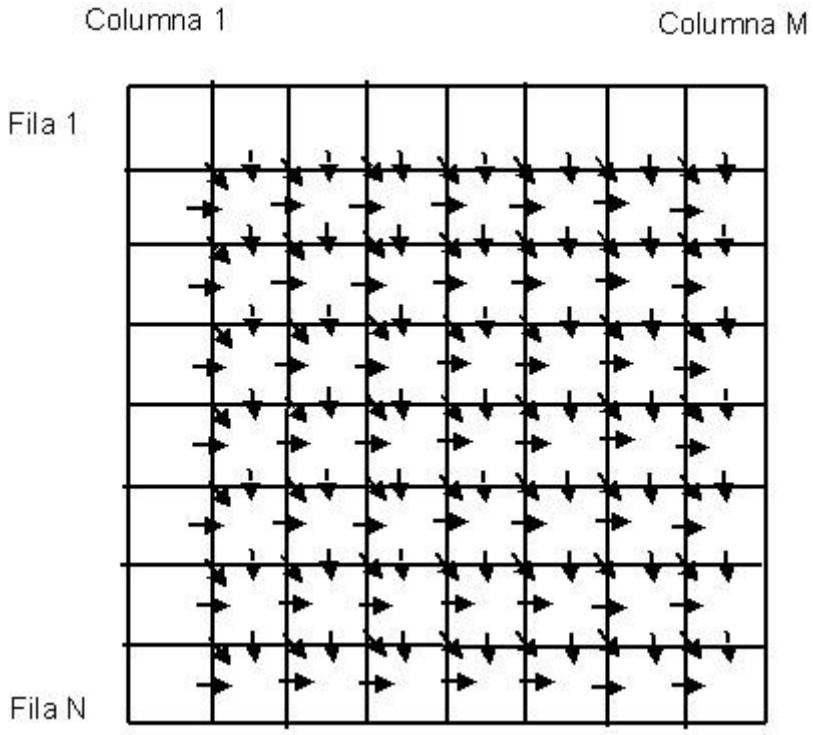


Figura 29 : Dependencias de datos de la matriz de puntuaciones

Siendo la figura anterior la matriz de puntuaciones y sus dependencias, esto nos impone una restricción en el momento de repartir el trabajo :

- Para calcular una nueva posición *Mi,j* de la matriz de puntuaciones hemos de haber calculado previamente $M_{i-1,j-1}$, $M_{i-1,j}$ y $M_{i,j-1}$ *(Figura 30)*

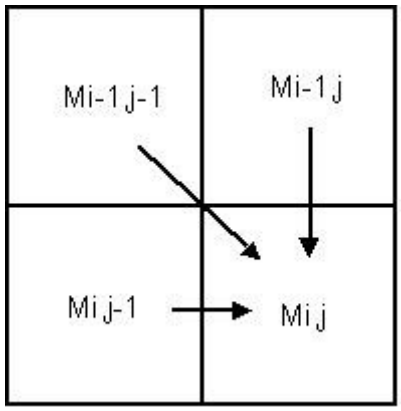


Figura 30 : Dependencia de datos para el calculo de un elemento de la matriz de puntuaciones

Debido a las dependencias de datos a la hora de calcular la matriz de puntuaciones se ha planteado un algoritmo basado en la estrategia de descomposición WaveFront.

Para poder paralelizar el bucle crítico necesitamos aseguramos que no haya dependencias entre iteraciones del bucle, la manera de conseguir esto es calculando las diagonales, que como podemos ver en la figura 31 es exactamente como funciona el WaveFront.[9]

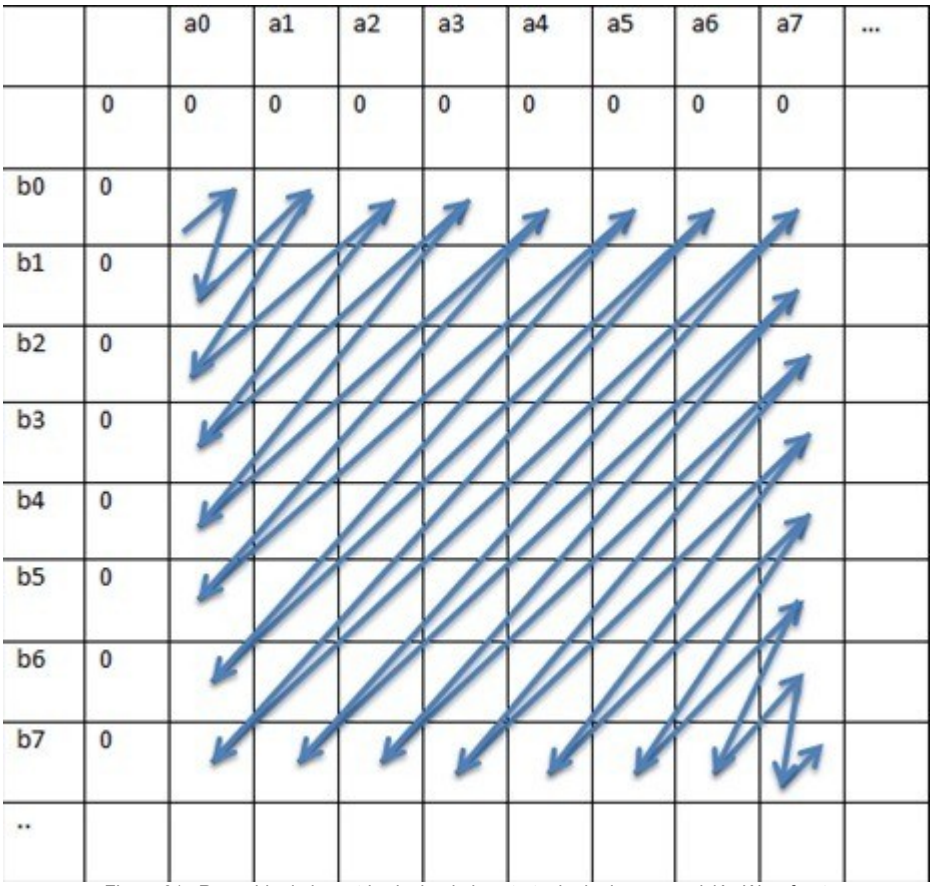


Figura 31 : Recorrido de la matriz siguiendo la estrategia de descomposición Wavefront

A la hora de repartir el trabajo primero se calcula el elemento (a0,b0). Seguidamente calculamos [(a0,b1), (a1,b0)] y estos 2 elementos ya podemos calcularlos en paralelo repartiendo un elemento a cada thread. Acto seguido, se calcula [(a0, b2), (a1,b1), (a2,b0)] el cual también podemos calcular en paralelo, y así hasta el final de la matriz.

En el algoritmo propuesto por nosotros a la hora de repartir el trabajo en vez de darle a cada thread un elemento de la diagonal de la matriz de puntuaciones, como se mostraba en la figura anterior, se le dará un bloque horizontal de elementos para que los calcule.

Cada uno de estos bloques tendrá un tamaño de :

$$M / nthreads.$$

De manera que si tenemos las secuencias :

secuencia 1 = *GAATTCAGTTAA*
secuencia 2 = *GGATCGAGTTGA*

Y queremos repartir el trabajo en 3 threads tendremos bloques de :

$$12 / 3 = 4 \text{ elementos}$$

El resultado obtenido sera una matriz de M columnas x N filas como muestra la figura 32

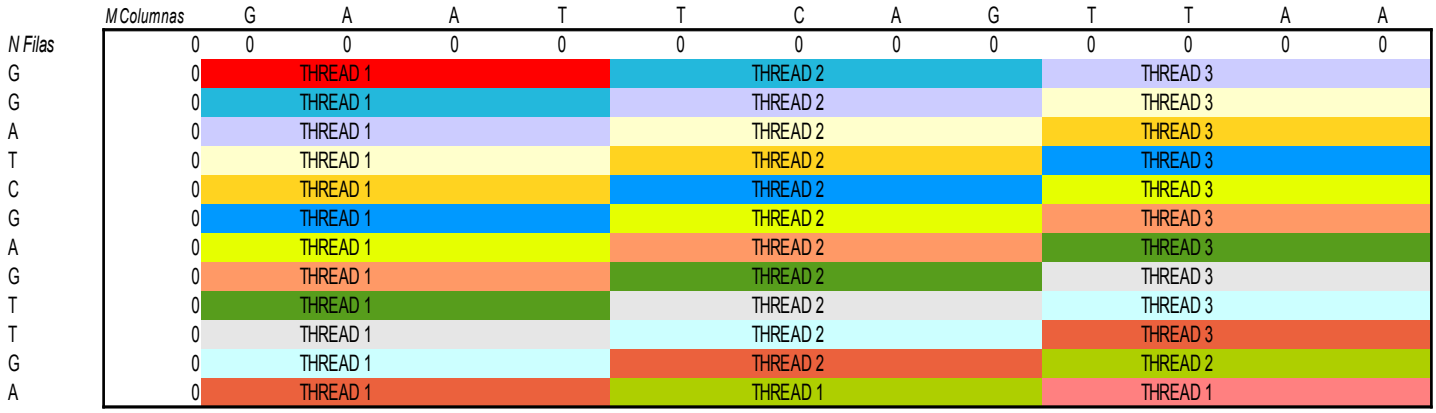


Figura 32: División de la matriz de puntuaciones en bloques

En la figura 33 podemos apreciar los bloques asignados a los threads en cada iteración.

Figura 33: Bloques asignados a los threads

Vemos en el ejemplo que se produce un desbalanceo de cómputo ya que sólo en el 71% de las iteraciones se ejecuta el número máximo de threads que tenemos definidos. En los siguientes apartados, entraremos más en detalle.

El algoritmo se ha implementado usando el lenguaje de programación C. A continuación se muestra el pseudocódigo *simplificado* de la función

Pseudocódigo

```
Accedemos a una diagonal de la matriz de puntuaciones en cada iteración
Para k = 2 ... k = numthreads+M

        Repartimos el trabajo dándole a cada thread un bloque de la diagonal
        #pragma omp parallel for default (shared) private(i)

        EN PARALELO :

        Cada thread ejecutará un bloque de la diagonal.  El primer thread
        ejecutará el bloque que se encuentre más a la  izquierda de la diagonal

        Cada thread calcula un bloque de p elementos.
                    Para p = 0 .. p<pfinal
                   {
                          Calcular_Elemento()

                   }
            }

        barrera de sincronización
```

Cómputo

En esta variante del algoritmo a cada thread le corresponde calcular un bloque, el cual esta formado por :

M / *nthreads* elementos de la matriz

Pero al principio y al final de la matriz de puntuaciones se ejecutan menos threads de los que hemos definido (desbalanceo de cómputo), este hecho es importante sobretodo si M (longitud de la secuencia 2) es pequeña ya que entonces el número de iteraciones en las que utilizamos el número máximo de threads definidos serán pocas.

Sincronización

En esta variante del algoritmo paralelo es necesario que todos los threads finalicen el cálculo de todos los elementos del bloque que tienen asignados antes de proseguir. Estos elementos son necesarios para poder calcular los elementos de la siguiente iteración.
Al coste de la sincronización además hay que añadirle que tendremos un tiempo de desbalanceo de carga ya que lo más probable es que un thread finalice su trabajo antes que otros. Por lo que los threads que han finalizado tendrán que esperar. Esta espera supone un coste en ciclos de reloj del procesador desaprovechados y por lo tanto un overhead.

Núm. veces que se sincroniza : nthreads + N -1 veces

Requerimientos de memoria

Los requerimientos de memoria no crecen respecto al número de threads ya que cada thread trabaja con variables locales. Concretamente cada thread tratará N / nthreads elementos de una fila de la matriz de puntuaciones cada vez
Los requerimientos de memoria globales no aumentan con el número de threads.

Memoria : N x M + (N + M)

Comunicación

En esta variante del algoritmo paralelo tenemos comunicación entre threads, ya que para poder calcular los elementos de la matriz de puntuaciones cada thread necesita leer las posiciones calculadas previamente en la iteración anterior.

La figura 34 representa los datos de entrada necesarios para calcular los elementos de un bloque de datos. Los datos que leemos de la fila superior han sido calculados en la iteración anterior (iteración i-1) por el mismo core que esta calculando ahora este bloque inferior de datos, por lo tanto los datos se encuentran en la memoria caché lo que comporta que no haya comunicación. En cambio, el elemento que se encuentra a la izquierda de nuestro bloque de datos ha sido calculado en otro thread que se encuentra en otro core, por lo tanto habrá comunicación entre cores para obtener ese dato (1 línea de cache).
El elemento que se encuentra diagonal al bloque de datos que estamos calculando comporta comunicación en la iteración i-1 para calcular el bloque de datos ejecutado por el thread t, pero en la iteración i no comporta comunicación debido a que ya tenemos en la caché este dato y no hay necesidad de pedírselo al thread t-1.

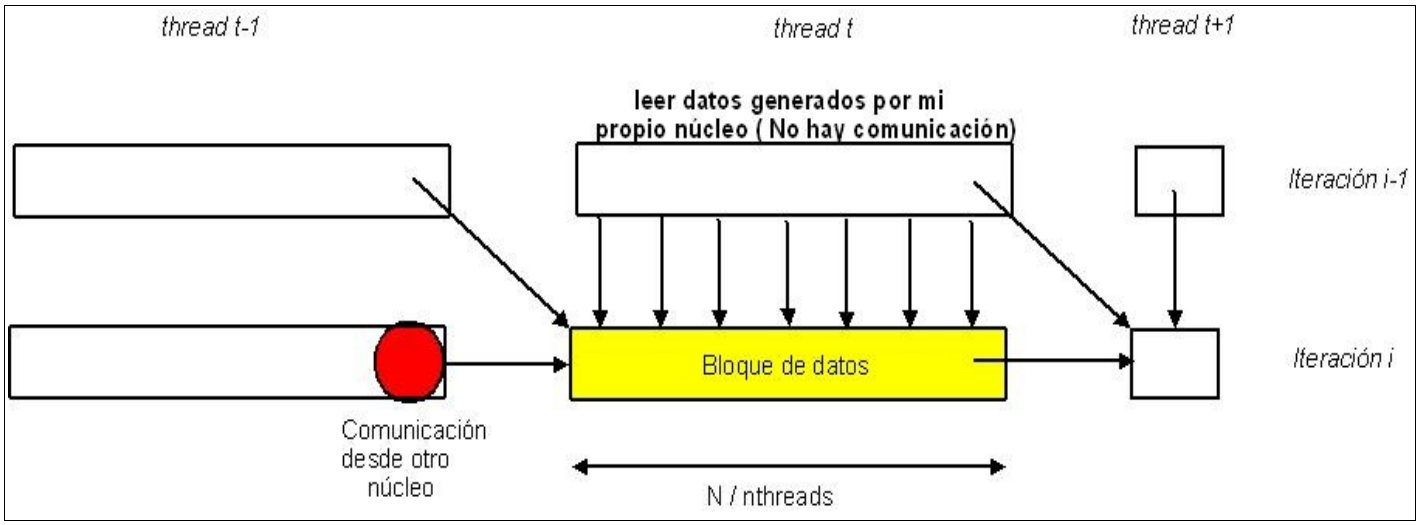


Figura 34: Comunicaciones que se producen en el cálculo de un bloque de datos

El volumen de datos leídos por cada thread, que previamente han sido escritos por otros threads se corresponde con 1 dato por iteración (1 línea de caché) entre vecinos.

Comunicación : 1 dato (1 linea de caché) x nthreads + N -1 veces

Diagnóstico

Si analizamos el algoritmo podemos apreciar que hay una gran cantidad de sincronización (M + numthreads sincronizaciones) ya que cada vez que calculamos la diagonal se tienen que sincronizar todos los threads. También apreciamos que a esta sincronización hay que sumarle un desbalanceo de carga en cada iteración ya que como hemos comentado los threads no acaban necesariamente todos a la vez, sino que suelen acabar en tiempos distintos y se tienen que esperar a que hayan finalizado todos.

También es un problema a tener en cuenta el desbalanceo de cómputo que se produce durante las primeras y ultimas iteraciones del cálculo de la matriz de puntuaciones, en las que no se aprovechan todos los threads que tenemos definidos ya que están trabajando menos threads de los que hemos asignado. Este es un problema derivado de tener que usar un método de descomposición como es el WaveFront para poder implementar el cálculo de la matriz de puntuaciones.

Por último tenemos un total de comunicaciones de nthreads + N -1 veces, que es el mismo número de veces que tenemos que sincronizar, pero por contra el coste de las comunicaciones en nivel de ciclos de ejecución es mucho menor que la sincronización (del orden de 10 veces inferior).

El diagnóstico que extraemos es el siguiente :

sincronización > comunicaciones > desbalanceo de cómputo

El problema más importante de nuestro programa es la sincronización ya que su coste es más elevado que las comunicaciones, mientras que el desbalanceo de cómputo es un problema que depende de N, cuando la N es grande el problema del desbalanceo decrece, mientras que el número de sincronizaciones aumenta, por lo tanto las optimizaciones que plantearemos intentarán minimizar la sincronización.

5.2.1. Optimizaciones

Se han implementado una serie de optimizaciones sobre la versión de grano fino con la finalidad de evitar el problema de la falsa compartición, que provoca falsa comunicación, y de reducir la sincronización entre threads a costa de aumentar el desbalanceo de cómputo. En este apartado explicaremos cómo funcionan, cuál es su objetivo y cómo han sido implementadas dentro de nuestro programa. La figura 35 muestra las 2 optimizaciones que realizaremos :

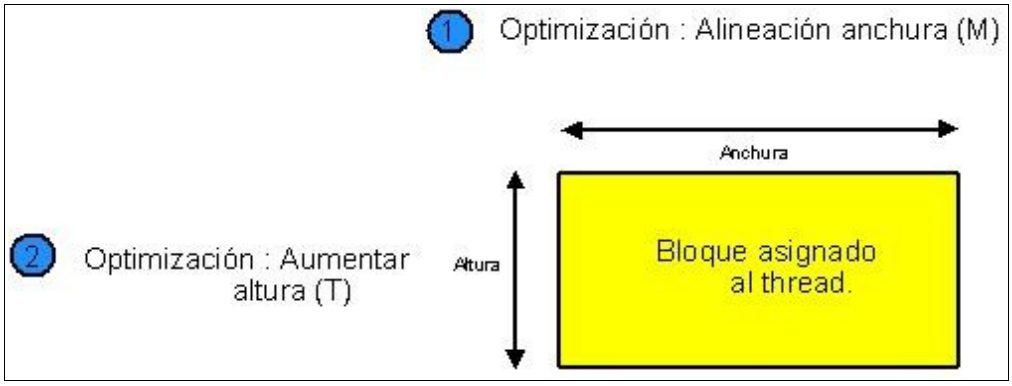


Figura 35: Esquema de las 2 optimizaciones que se realizarán

1. Alineación del bloque de datos asignado a cada thread

La memoria caché está estructurado por celdas, donde cada celda almacena un byte. La entidad básica de almacenamiento la conforman las filas, llamadas también líneas de caché, que por lo regular son de 64 bytes, con las direcciones de 0 a 63 se accede a la línea de cache 0, con las direcciones 64 a 127 a la línea de cache 1, etc.

Cuando se copia o se escribe información de la RAM a caché, por cada movimiento siempre cubre una línea de caché. Las líneas de caché de uso más intensivo se mantienen en una caché de alta velocidad situada dentro de la CPU. Cuando el programa necesita leer una palabra de memoria, el hardware de caché determina si la línea necesaria esta o no en la caché

El objetivo de esta optimización es asegurar que la partición en columnas no corte una línea de caché, porque se pueden generar problemas de Falsa compartición.
La Falsa compartición ocurre cuando dos o más núcleos están actualizando bytes de memoria distintos que coinciden en la misma línea de caché, esto provoca falsa comunicación.

Esta optimización la hemos llevado a cabo comprobando que la dirección de una línea de datos horizontal asignada a cada thread sea múltiplo de 64 bytes. En caso de no serlo, aumentamos el tamaño de los datos que cogemos hasta que lo sea.

Se ha generado un diagrama de flujo para representar gráficamente cómo hemos implementado esta optimización *(Figura 36)*

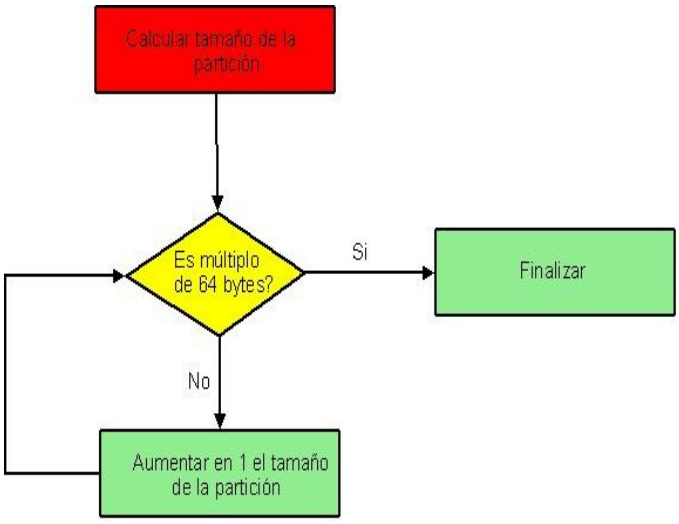


Figura 36: Diagrama de flujo perteneciente a la optimización de alineación de bloques de datos asignado a cada thread

2. Variación de la altura de los bloques de datos asignados a cada thread

El objetivo principal de esta optimización es reducir el overhead de sincronización.

La manera de reducir la sincronización del programa es aumentando la altura de los bloques de datos que se calculan en la diagonal. De manera que el número de iteraciones que haga el programa sea inferior y por lo tanto se reduzca el número de veces que se tienen que sincronizar los threads.

La diferencia ahora es que en la versión anterior de nuestro algoritmo la altura de los bloques era 1 (una fila). Ahora para realizar esta optimización se ha definido un parámetro extra T, el cual nos permitirá definir la altura de los bloques.

De manera que si tenemos las mismas secuencias que en el ejemplo anterior :

secuencia 1 = GAATTCAGTTAA
secuencia 2 = GGATCGAGTTGA

Y queremos repartir el trabajo en 3 threads tendremos bloques de :

12 / 3 = 4 elementos

Si definimos la T = 3. El resultado obtenido sera el de la figura 37

M Columnas		G	A	A	T	T	C	A	G	T	T	A	A
N Filas	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	THREAD 1					THREAD 2				THREAD 3		
G	0	THREAD 1					THREAD 2				THREAD 3		
A	0	THREAD 1					THREAD 2				THREAD 3		
T	0	THREAD 1					THREAD 2				THREAD 3		
C	0	THREAD 1					THREAD 2				THREAD 3		
G	0	THREAD 1					THREAD 2				THREAD 3		
A	0	THREAD 1					THREAD 2				THREAD 2		
G	0	THREAD 1					THREAD 2				THREAD 2		
T	0	THREAD 1					THREAD 2				THREAD 2		
T	0	THREAD 1					THREAD 1				THREAD 1		
G	0	THREAD 1					THREAD 1				THREAD 1		
A	0	THREAD 1					THREAD 1				THREAD 1		

Figura 37: División de la matriz de puntuaciones en bloques de altura T

En la figura 38 podemos apreciar los bloques asignados a los threads en cada iteración.

Iteración 1	Thread 1
Iteración 2	Thread 1 y 2
Iteración 3	Thread 1, 2 y 3
Iteración 4	Thread 1, 2 y 3
Iteración 5	Thread 1 y 2
Iteración 6	Thread 1

Figura 38: Bloques asignados a los threads

Vemos como hemos empeorado el desbalanceo de cómputo en relación a la versión anterior ya que en el ejemplo del apartado anterior teniamos un 71% de iteraciones en las que se ejecutaban el número máximo de threads. Por contra ahora tan sólo en el 33% de las iteraciones se ejecuta el número máximo de threads.
En el dibujo también se puede apreciar como hemos reducido el número de iteraciones (sincronizaciones) de 14 que realizaba la versión anterior a 6. Ahora el volumen de datos escrito entre sincronizaciones por cada thread es de :

*(M / nthreads) * T elementos*

El volumen de datos leídos por cada thread, que previamente han sido escritos por otros threads (comunicación) se corresponde a :

T elementos

Vemos como esta optimización ha aumentado el número de cálculos que hacemos en paralelo entre sincronizaciones de *(núm. columnas / thread)* a *(núm. columnas / thread) * T.*
La cantidad de comunicación entre threads ha aumentado en volumen de datos que leemos, pero ha disminuido en la cantidad global de veces que cada thread tiene que leer datos.

A continuación se muestra el pseudocódigo *simplificado* de la optimización

Pseudocódigo

Accedemos a la diagonal correspondiente a cada iteración, donde cada bloque que la forma tiene una longitud (num.columnas / thread) y un anchura T

```
Para k = 2 ... k = (numthreads+(M/T))*T
```

Repartimos el trabajo dándole a cada thread un bloque de la diagonal

```
#pragma omp parallel for default (shared) private(i)
```

EN PARALELO :

Cada thread ejecutará un bloque de la diagonal.
El primer thread ejecutará el bloque que se encuentre más a la izquierda de la diagonal.
Esta vez hay que decrementar T elementos el indice para posicionarnos en el bloque siguiente.

```
                i = i - T

                Recorremos las T filas que forman un bloque
                Para j =0 .. j < T
                {

                 y en cada fila calculamos p elementos.
                    Para p = 0 .. p<pfinal
                    {

                         Calcular_Elemento()
                     }
                 }
      }
```

```
barrera de sincronización
```

5.3. Experimentación

En este apartado se describirán los procesadores utilizados. A continuación se presentará el método experimental para tomar medidas de rendimiento de cada procesador y los workloads utilizados.

5.3.1. Sistemas de cómputo

Para cada procesador se muestra el número de procesadores, cuantos cores tiene cada procesador y cuantos threads puede ejecutar cada uno.

Intel Pentium D	
Procesador	Intel Core 2 Duo @ 3.00 GHz
Cores	2
Threads per core	1

Intel Pentium Core 2 Duo	
Procesador	Intel T8100 Duo @ 2,1 GHz
Cores por procesador	2
Threads per core	1

Sun UltraSPARC T2	
Procesador	SUN SPARC-E-T5120, UltraSPARC T2 @ 1165 MHz
Cores por procesador	4
Threads per core	8

5.3.2. Métodos de medición y análisis

Para medir tiempos de la aplicación paralela en el procesador Sun UltraSPARC hemos utilizado la función *gettimeofday.*

Para medir tiempos en procesadores x86 hemos utilizado una instrucción maquina especial.
Estas instrucciones hacen uso del acumulador principal EAX y del secundario EDX de la arquitectura Intel x86 para obtener el número de ciclos totales de cada uno de los threads.

5.3.3. Métricas y Workloads

Igual que en la experimentación del apartado 4.4 la métrica fundamental que se ha medido en las ejecuciones de ambas variantes ha sido el tiempo de ejecución del programa.

En las gráficas el eje Y muestra el Speedup mientras que el eje X están expuestos los 3 sistemas sobre los que hemos probado el programa. El Speedup se calcula haciendo t^s / t^0 donde t^s es el tiempo que se requiere para ejecutar el programa serie y t^0 es el tiempo que se requiere para ejecutar el programa paralelo

Los programa usado para medir los tiempos son:

- Para la versión large – grained se utiliza el código en serie con todas las optimizaciones, además de la función explicada en el apartado dedicado al large-Grain, que es la encargada de repartir el trabajo entre los threads.
- Para la versión fine-grained se utiliza la versión con todas las optimizaciones discutidas en el apartado dedicado a la paralelización fine-grained.

Se han hecho las pruebas alineando más de una secuencia y con tamaños diferentes, en todos los casos el número de iteraciones que tiene que hacer el programa es el mismo, pero como se podrá apreciar los resultados varían dependiendo del tamaño de las secuencias. Las pruebas para ambos casos se han realizado con 1 y 2 threads respectivamente en las arquitecturas x86 y con 1 a 32 threads en la arquitectura UltraSparc.

El número de tamaños y secuencias usadas para la versión large-grain son :

- Alineamos 2 secuencias de longitud 10.000 entre ellas.
- Alineamos 100 secuencias de longitud 200 entre ellas.
- Alineamos 400 secuencias de longitud 50 entre ellas.

El número de tamaños y secuencias usadas para la versión fine-grain son :

- Alineamos 2 secuencias de longitud 10.000 entre ellas.

Estos workload se escogen porque, cómo se verá más adelante, representan casos significativos.

En la versión fine-grain también variaremos el tamaño de la altura de bloques de datos, para poder encontrar cual es el tamaño que nos permite minimizar más la comunicación entre threads y aumentar el cálculo de datos en paralelo. Los tamaños elegidos son :

- Tamaño = 50 filas.
- Tamaño = 100 filas.
- Tamaño = 500 filas.

5.4. Resultados

Los resultados del estudio se dividen en tres apartados:

En el primero se verán y discutirán los resultados de la paralelización large-grained.
En el segundo se verán y discutirán los resultados para la versión fine-grained. Finalmente en el ultimo apartado se compararan las dos versiones y se analizará cual es mejor dependiendo de la situación en que nos encontremos.

5.4.1. Resultados en las 3 arquitecturas de la paralelización large-grained

En la figura 39 se muestran el speedup obtenido de ejecutar el programa sobre los 3 sistemas usando 1 thread.

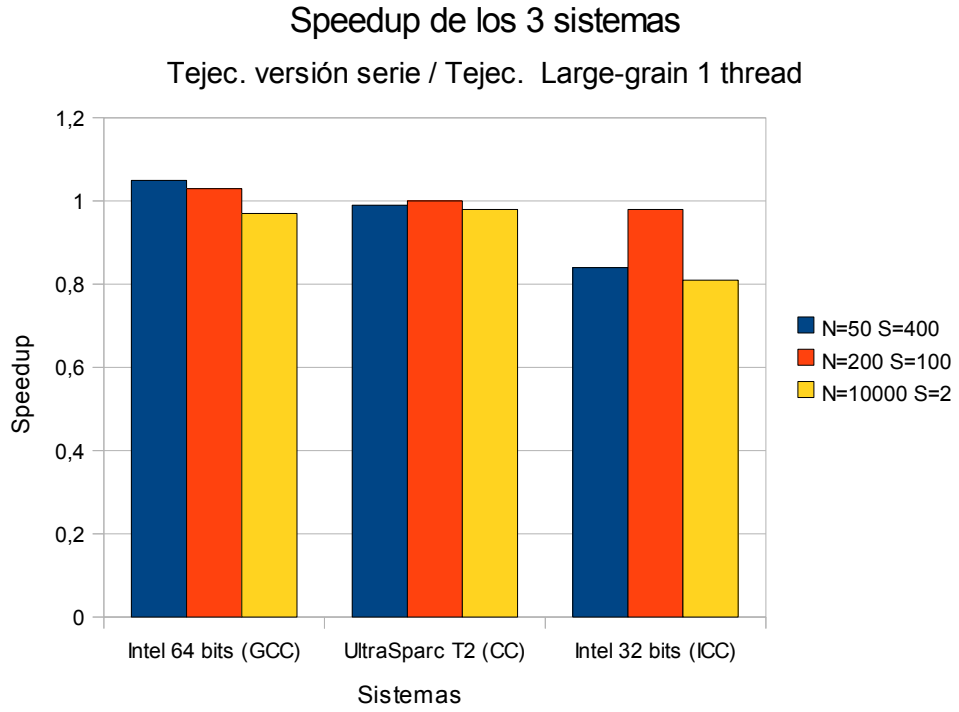


Figura 39: Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela large-grain usando 1 thread en cada uno de los procesadores (Speedup)

El speedup obtenido al dividir el tiempo de ejecución de la versión serie entre el tiempo de ejecución de la versión paralela large-grain con 1 thread nos muestra:

- La versión de 64 bits y UltraSPARC funcionan bien ya que alcanzamos un speedup cercano al 1. Es decir, la variante con 1 thread tarda más o menos lo mismo en ejecutarse que la versión serie.

- La versión de 32 bits funciona mal en relación a la versión serie ya que tan sólo en el caso de N=200 y S =100 su tiempo de ejecución se acerca a la versión serie.

En la figura 40 se muestran el speedup obtenido de ejecutar el programa sobre los 3 sistemas usando 2 threads.

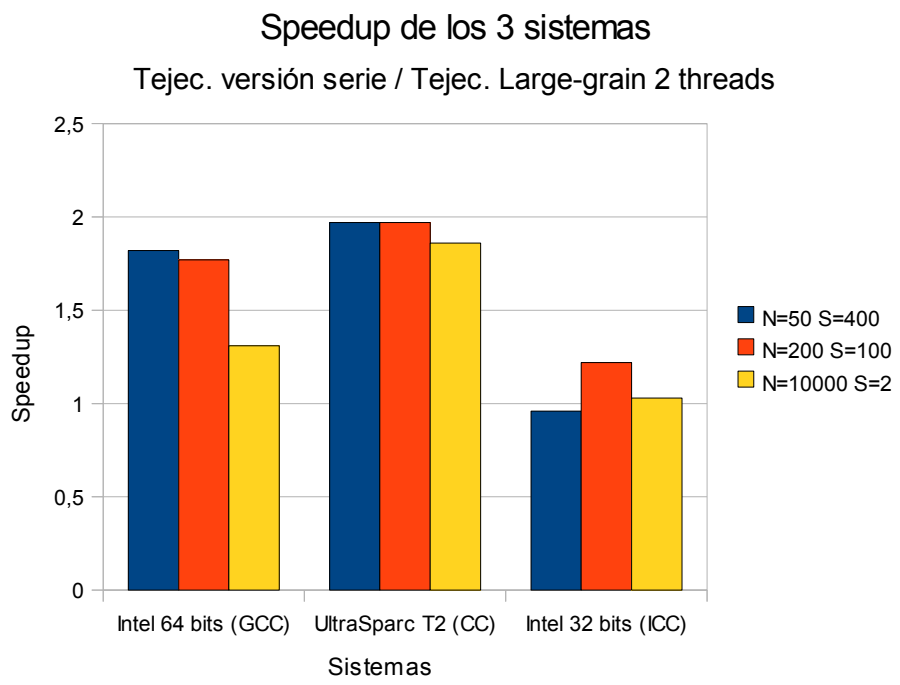


Figura 40 Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela large-grain usando 2 threads en cada uno de los procesadores (Speedup)

El procesador Intel de 64 bits dispone de dos núcleos. En cada uno de ellos es capaz de ejecutar un thread. Cuando ejecutamos el algoritmo de grano grueso con dos threads y el Workload (N = 50, S=400) obtenemos un speedup de 1.82 y una eficiencia por thread del 85%. Mientras que para el workload (N=200, S=100) obtenemos un speedup del 1.77 y una eficiencia por thread del 85%. Finalmente si ejecutamos el programa con el workload (N=10.000, S= 2) obtenemos un speedup del 1.31 y una eficiencia por thread del 52%

Vemos que en el procesador Intel de 64 bits a medida que vamos aumentando el tamaño de la secuencia vamos perdiendo eficiencia por thread. Por lo tanto, podemos decir que el multithreading de granularidad gruesa en el procesador de 64 bits es bueno al trabajar con secuencias no muy grandes ya que tenemos un speedup entorno al 1,8. En cambio a medida que va aumentando el tamaño de las secuencias empeora la eficiencia por thread, seguramente debido a que para tratar secuencias grandes los datos no caben en la memoria caché del procesador, y hay que obtenerlos de la memoria principal. También es muy probable que haya conflicto entre los threads / cores al acceder a la memoria principal por un canal compartido.

En el procesador UltraSPARC disponemos de un procesador con cuatro núcleos (cores). A diferencia de los nodos anteriores, cada uno de estos cores puede ejecutar hasta 8 threads en paralelo. Al ejecutar el algoritmo de grano grueso con dos threads obtenemos un speedup cercano al 2 para los Workloads (N=50, S=400) y (N =200, S=100), mientras que para el Workload (N=10.000, S=2) obtenemos un speedup del 1.86

Por lo tanto, podemos decir que el multithreading de granularidad gruesa en el procesador UltraSparcT2 es bueno en todos los casos ya que al ejecutar el programa con 2 threads se ha conseguido reducir casi al 50% el tiempo de ejecución en los 3 casos. Por tanto, no se satura el ancho de banda.

El procesador Intel de 32 bits dispone de dos núcleos. En cada uno de ellos es capaz de ejecutar un thread. Cuando ejecutamos el algoritmo de grano grueso con dos threads obtenemos y el Workload (N = 50 y S=400) obtenemos un speedup de 0.96. Si ejecutamos el programa con el Workload (N=200 y S=100) obtenemos un speedup de 1.22 y finalmente con el Workload (N=10.000 y S= 2) obtenemos un speedup de 1.03 en relación a la versión serie

Si comparamos la versión large-grain en el procesador de 32 bits al ejecutarla con 1 thread y con 2 threads, vemos que el tiempo de ejecución se reduce aproximadamente un 30%. Pero todo y haberse reducido el tiempo de ejecución esta versión apenas funciona ligeramente mejor que la versión serie con excepción del workload (N=50 S=400) que sigue funcionando peor que la versión serie.

Por lo tanto, podemos concluir que el multithreading de granularidad gruesa en el procesador de 32 bits NO es ventajoso ya que al aumentar el número de threads el tiempo de ejecución de esta variante sigue siendo superior a la versión serie.

A continuación compararemos el tiempo de ejecución en forma de ns. / Complejidad cuando ejecutamos la aplicación large-grain variando el número de threads desde 1 hasta 32 *(Figura 41)*. Esto nos servirá para determinar cual es el número de threads idóneo para nuestro programa en la maquina UltraSPARC.

Con el Workload (N=10.000, S = 2) tan solo podemos ejecutar la aplicación con 4 threads ya que con 2 secuencias sólo son posibles 4 alineamientos. Si probamos de ejecutar la aplicación aumentando el número de secuencias de manera que cada thread pueda realizar un alineamiento nos encontramos con un problema de memoria, ya que se supera el tamaño de memoria disponible debido a que tenemos una matriz de 10.000 x 10.000 x 4 bytes por cada thread.
Por lo tanto en los apartados correspondientes a 8,16 y 32 threads tan sólo se ha medido el tiempo de los Workloads (N=50, S=400) y (N=200,S=100)

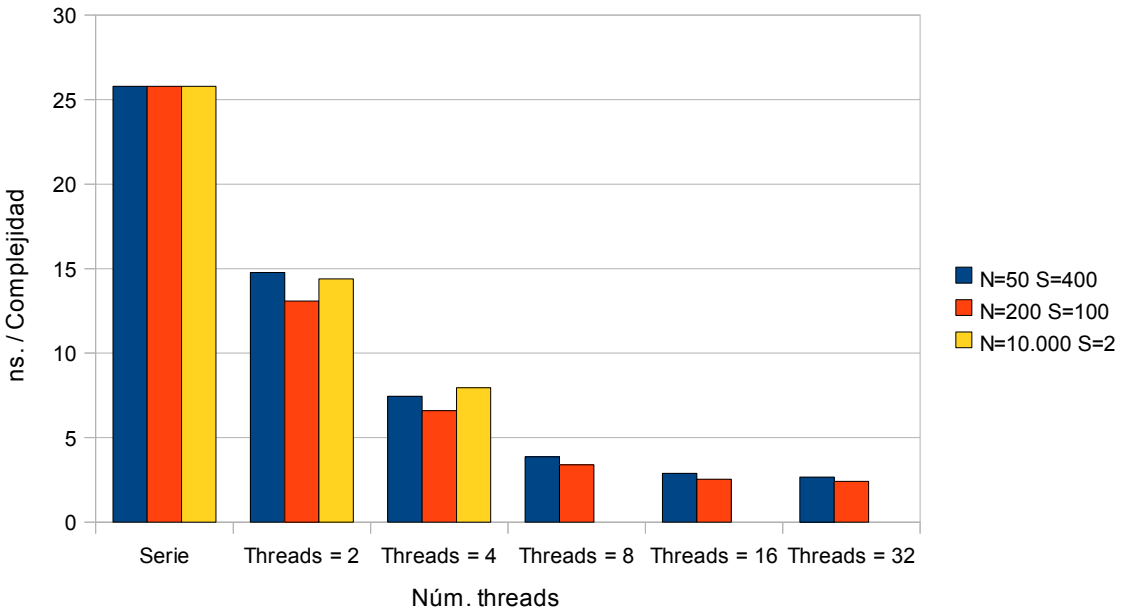


Figura 41: Tiempo de ejecución divididos por complejidad, para la ejecución del algoritmo paralelo large-grain en el procesador SPARC variando el número de threads

Al aumentar el número de threads con el que ejecutamos la aplicación paralela se va perdiendo eficiencia por thread.

En el procesador UltraSparc con el Workload (N=50, S = 400) al ejecutar el algoritmo con 2, 4 y 8 threads obtenemos una eficiencia por thread cercana al 100%, al ejecutar con 16 threads la eficiencia por thread es del 51% y finalmente al ejecutar con 32 threads la eficiencia por thread es del 25%.

Al ejecutar el Workload (N=200, S=100), al igual que en el caso anterior obtenemos una eficiencia por thread cercana al 100% cuando ejecutamos la aplicación con 2, 4 y 8 threads,al ejecutar con 16 threads la eficiencia por thread baja hasta el 50% y finalmente al ejecutar con 32 threads la eficiencia por thread es del 20%.

Con el Workload (N=10.000, S = 2) tan sólo podemos ejecutar la aplicación con 2 y 4 threads, ya que cómo hemos comentado antes si aumentamos el número de threads superamos el máximo de memoria del sistema. El ejecutar este Workload con 2 y 4 threads obtenemos una eficiencia por thread del 100%:

A continuación compararemos el rendimiento obtenido al ejecutar el algoritmo large-grain en los distintos procesadores que estamos analizando. En la figura 42, se muestra una gráfica resumen de los tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo de grano-fino, en los tres sistemas analizados (utilizando el máximo número de threads posible por procesador). En la figura 43 se muestra una gráfica resumen del Speedup.

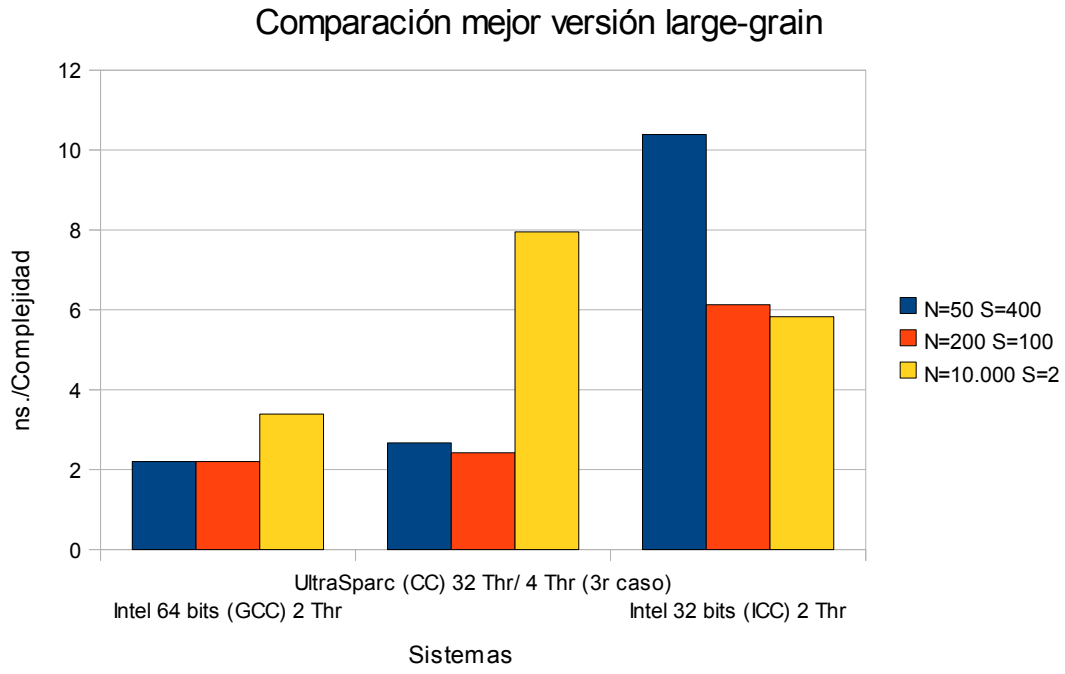


Figura 42: Tiempo de ejecución divididos por complejidad, para la ejecución del algoritmo paralelo large-grain en los 3 procesadores con el máximo número de threads

Como podemos apreciar en la gráfica con el procesador Intel de 64 bits es con el que obtenemos el mejor rendimiento en los 3 Workloads. El procesador UltraSPARC también nos da un rendimiento bueno con 32 threads acercándose bastante al tiempo del Intel de 64 bits.

Hay que tener en cuenta que con el workload (N=10.000 S=2) no podemos utilizar todo el potencial de la máquina UltraSPARC, ya que sólo podemos utilizar 4 threads debido a que como hemos visto anteriormente si usamos más threads se supera el tamaño de la memoria disponible.

Este problema se resolverá con la versión Fine-grained que veremos a continuación.

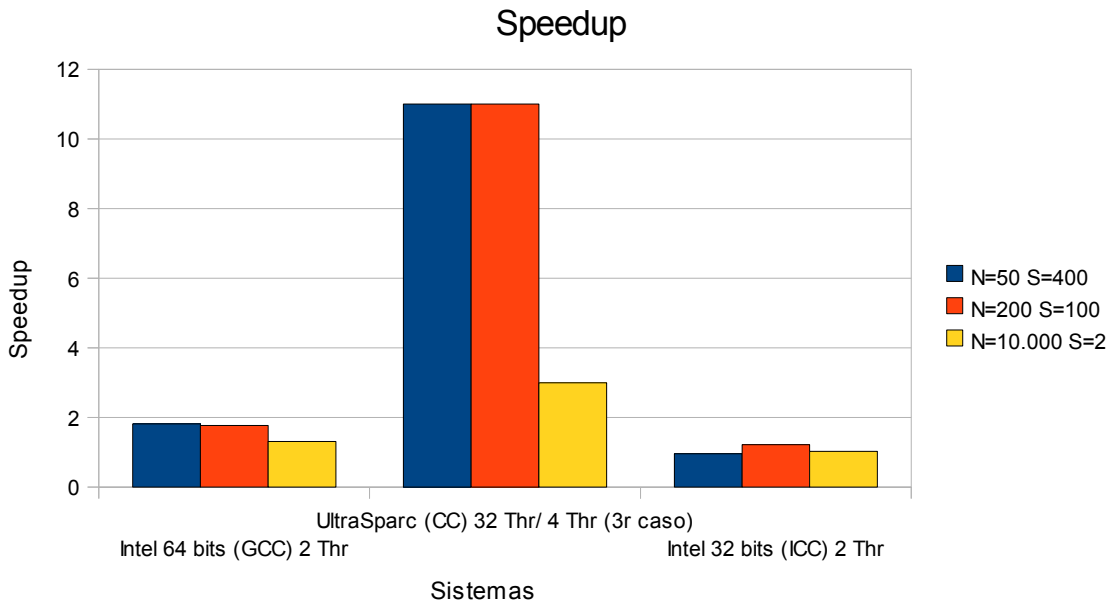


Figura 43: Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela large-grain usando el máximo número de threads en cada procesador.

El procesador que obtiene un mayor speedup al ejecutar el programa con el número máximo de threads per core es el UltraSPARC, el cual obtiene un speedup de 11 al ejecutarla con 32 threads frente al 1.8 del Intel de 64 bits y 1.22 del Intel de 32 bits. La razón de que la UltraSPARC obtenga un speedup tan elevado es debido a que el rendimiento de éste con 1 thread es muy malo.

5.4.2. Resultados en las 3 arquitecturas de la paralelización fine-grained

En esta variante del algoritmo paralelo se ha definido un parámetro extra T, el cual nos permitirá definir la altura de los bloques, en este apartado se ha ido variando el tamaño de la altura de bloques de datos, para poder encontrar cual es el tamaño que nos da un mejor resultado.

El hecho de aumentar la T nos permite disminuir el número de sincronizaciones que realizamos para calcular la matriz de puntuaciones pero aumenta el desbalanceo de cómputo, ya que cómo hemos visto en el ejemplo cuanto más aumenta el valor de la T más disminuye el número de iteraciones en las que usamos el número máximo de threads .

El hecho de disminuir la T, es el caso opuesto, nos permite disminuir el desbalanceo de cómputo pero por contra aumenta el número de veces que los threads tienen que sincronizarse ya que aumenta el número de iteraciones que tiene que realizar el algoritmo para calcular la matriz de puntuaciones.

En la figura 44 se muestran el Speedup obtenido de ejecutar el programa con el Workload (N=10.000 y S=2) sobre los 3 procesadores usando 1 thread:

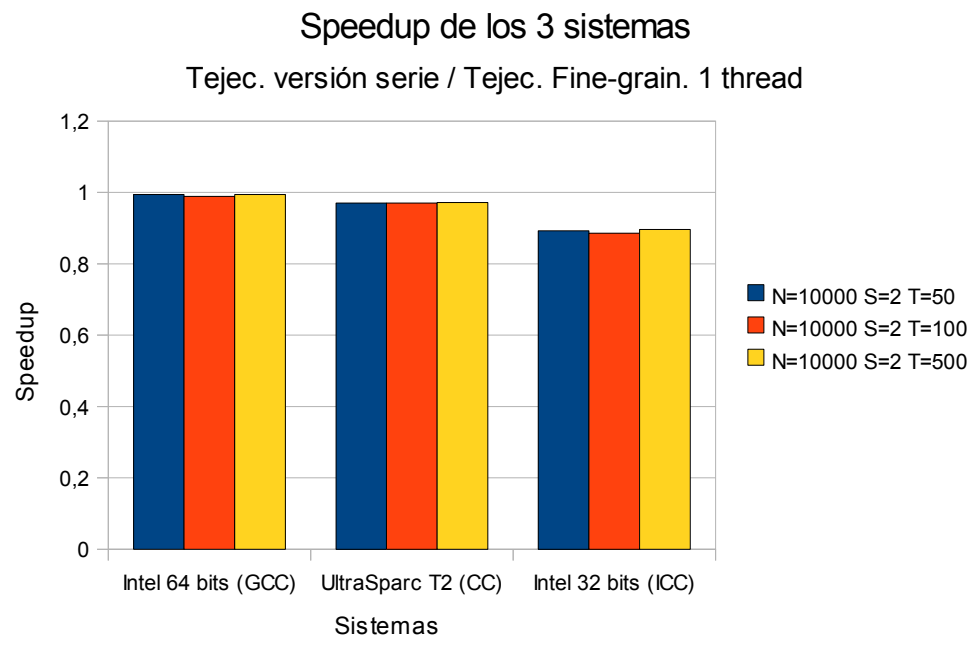


Figura 44: Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela fine-grain usando 1 thread en cada uno de los procesadores (Speedup)

Los resultados obtenidos son parecido a la versión large-grained, la versión de 64 bits y UltraSPARC funcionan bien ya que alcanzamos un speedup cercano al 1, mientras que la versión de 32 bits funciona mal en relación a la versión serie ya que se tarda más tiempo en ejecutarse.

En la figura 45 se muestran el Speedup obtenido de ejecutar el programa con el Workload (N=10.000 y S=2) sobre los 3 procesadores usando 2 threads :

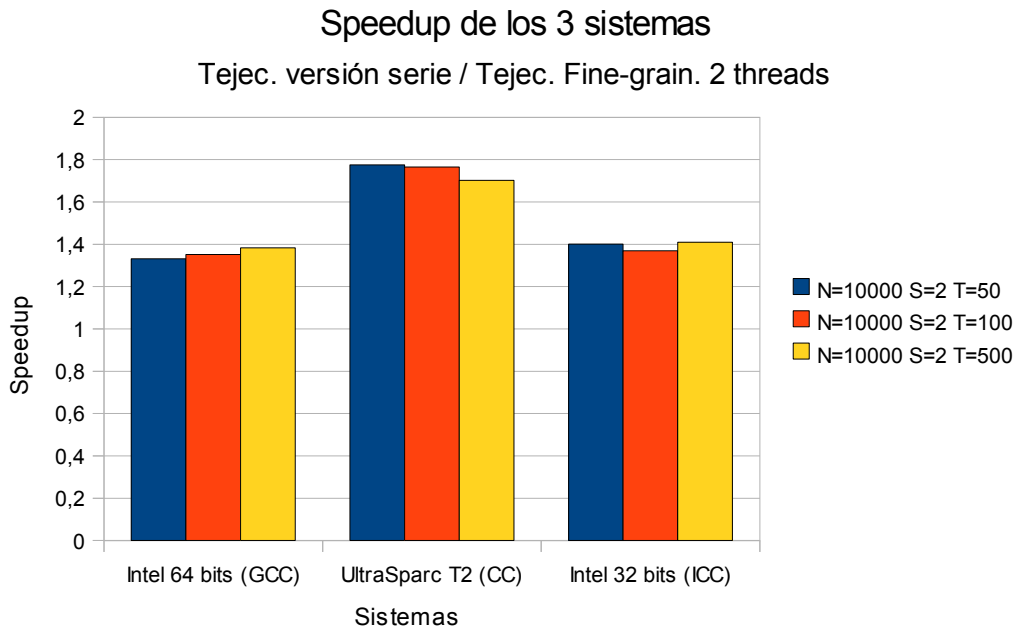


Figura 45: Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela fine-grain usando 2 threads en cada uno de los procesadores (Speedup)

Como podemos apreciar en las gráficas del speedup, el mejor valor de T depende del procesador con el que realicemos las mediciones. En el Intel de 32 y 64 bits el mejor speedup lo obtenemos con T = 500 mientras que en la UltraSPARC con T = 50.

En el procesador Intel 64 bits disponemos de un procesador con dos núcleos (cores). Cada uno de estos cores es capaz de ejecutar un thread. Al ejecutar el algoritmo paralelo con dos threads y el Workload (N = 10.000, S=2, T=50) obtenemos un speedup de 1.33 y una eficiencia por thread del 51%. Si ejecutamos el programa con el Workload (N = 10.000, S=2, T=100) obtenemos un speedup de 1.35 en relación a la versión serie y una eficiencia por thread del 53%. Finalmente con el Workload (N = 10.000, S=2, T=500) obtenemos un speedup de 1.38 y una eficiencia por thread del 57%

En la figura 44 se puede apreciar como con el workload (N=10.000 S=2 T=500) el speedup obtenido es 1.38, éste ya es superior a su homólogo de la versión de grano grueso que era de 1.31. Si se hicieran pruebas alineando pocas secuencias de tamaño cada vez mayor, se podría apreciar como cada vez sería más significativa la mejoría de la versión de grano fino por contra de la versión de grano grueso.

En el procesador UltraSPARC T2 cuando ejecutamos con 2 threads obtenemos que para (N = 10.000, S=2, T=50), (N = 10.000, S=2, T=100) y (N = 10.000, S=2, T=500) el tiempo de ejecución se reduce a la mitad. Obtenemos un speedup cercano al 1.8 en todos los casos. La eficiencia por thread es del 90%.

En el procesador Intel 32 bits, disponemos de un procesador con dos núcleos (cores). Cada uno de estos cores es capaz de ejecutar un thread. Al ejecutar el algoritmo paralelo de grano fino con dos threads y los Workloads (N = 10.000, S=2, T=50) y (N = 10.000, S=2, T=500) obtenemos un speedup de 1.40 y una eficiencia por thread del 73%. Mientras que con el workload (N = 10.000, S=2, T=100) se obtiene un speedup de 1.37 y una eficiencia por thread del 71%

A continuación compararemos el tiempo de ejecución en forma de ns. / Complejidad cuando ejecutamos la aplicación large-grain variando el número de threads desde 1 hasta 32 *(Figura 46).* Esto nos servirá para determinar cual es el número de threads idóneo para nuestro programa en la maquina UltraSPARC. El Workload elegido para realizar la prueba es (N = 10.000, S=2, T=50), que como hemos podido comprobar es con el que conseguimos el mejor Speedup en el procesador UltraSparc T2.

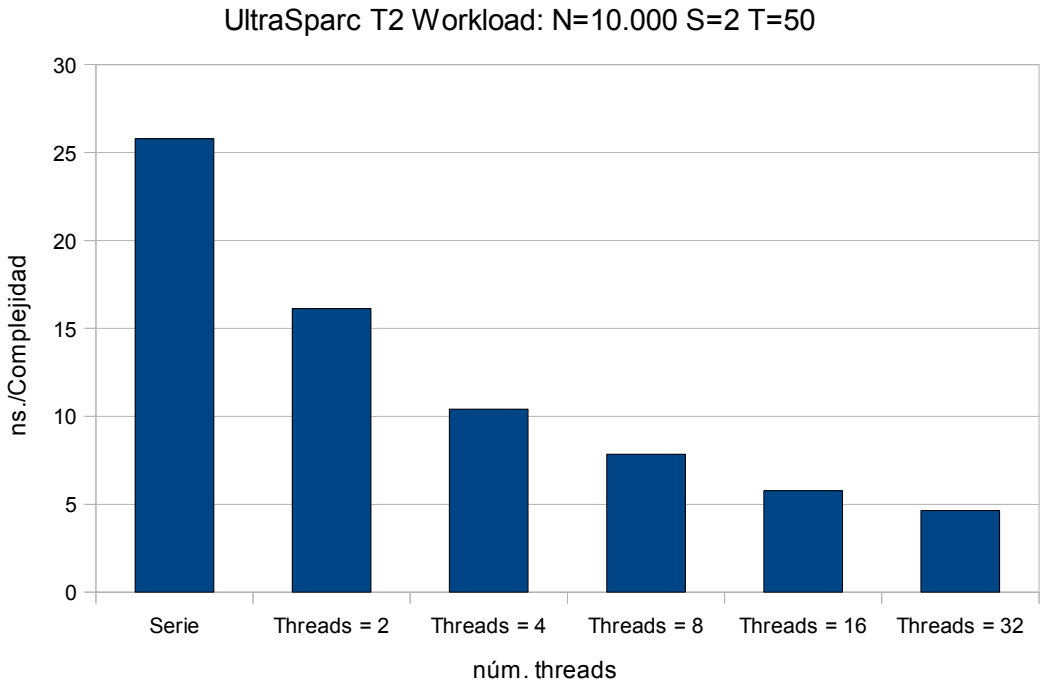


Figura 46: Tiempo de ejecución divididos por complejidad, para la ejecución del algoritmo paralelo fine-grain en el procesador SPARC variando el número de threads

Al aumentar el número de threads con el que ejecutamos la aplicación paralela se va perdiendo eficiencia por thread. Con 2 threads obtenemos una eficiencia por thread del 85%. Al ejecutar la aplicación con 4 threads la eficiencia por thread es del 72%. Con 8 threads la eficiencia por thread se reduce hasta el 50%. Con 16 threads tenemos una eficiencia por thread del 44% y finalmente con 32 threads la eficiencia por thread es del 39%.

En concreto, con hasta cuatro threads, se ejecuta un thread por core. Al ir añadiendo threads por core, se va perdiendo cada vez mas eficiencia.

A continuación comparemos el rendimiento obtenido al ejecutar el algoritmo paralelo en los distintos procesadores que estamos analizando. En la figura 47, se muestra una gráfica resumen de los tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo de grano-fino, en los tres sistemas analizados (utilizando el máximo número de threads posible por procesador). En la figura 48 se muestra una gráfica resumen del Speedup.

El Workload utilizado para estas gráficas es (N = 10.000 S=2) con la mejor T en para cada uno de los casos. Es decir en el Intel de 32 y 64 bits T = 500 mientras que en la UltraSPARC T = 50

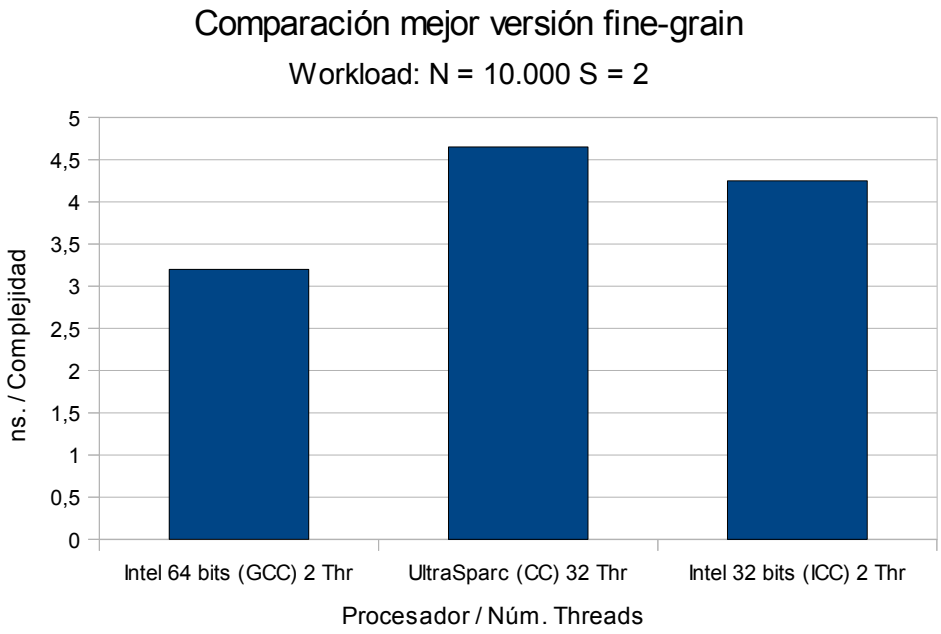


Figura 47: Tiempo de ejecución divididos por complejidad, para la ejecución del algoritmo paralelo fine-grain en los 3 procesadores con el máximo número de threads

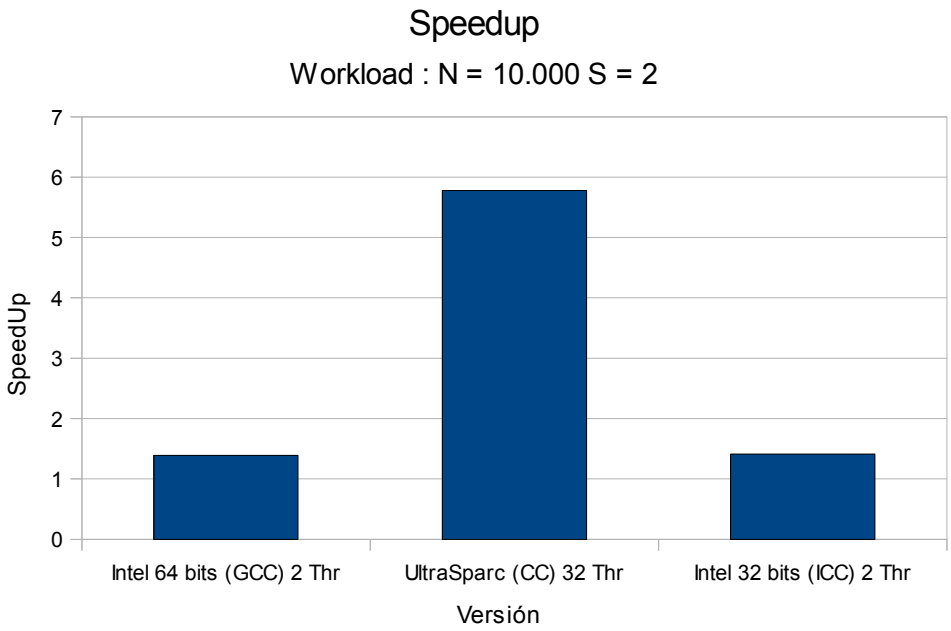


Figura 48: Tiempo de ejecución de la versión serie dividido por el tiempo de ejecución de la versión paralela fine-grain usando el máximo número de threads en cada procesador.

Como podemos apreciar en la primera gráfica con el procesador Intel de 64 bits es con el que obtenemos el mejor rendimiento con un speedup de 1.3 frente al procesador Intel de 32 bits y un speedup de 1.4 frente al UltraSPARC.

Por contra, el procesador que obtiene un mayor speedup al ejecutarla con el número máximo de threads per core es el UltraSPARC, el cual obtiene un speedup de 5.8 al ejecutarla con 32 threads frente al 1.3 del Intel de 64 bits y 1.4 del Intel de 32 bits. Como se ha comentado en el apartado anterior la razón de que el UltraSPARC obtenga un speedup tan elevado en comparación al resto de sistemas es debido a que el rendimiento de éste con 1 thread es muy malo.

5.4.3. Conclusiones de la paralelización large-grained y fine-grained

En la figura 49 y figura 50, se muestran 2 gráficas resumen de los tiempos de ejecución divididos por complejidad, para la ejecución del algoritmo de grano grueso y el de grano-fino, en los tres sistemas analizados.

Para la versión de grano grueso se ejecuta el programa con el número de threads que mejores resultados nos ha dado en las pruebas para cada uno de los procesadores, mientras que para la versión de grano fino se ejecuta el programa con el mejor número de threads y con el mejor valor de T para cada uno de los procesadores.

Con el Workload (N=1.000, S=20) los valores de T usados son para el sistema x86-64 y x86-32 T=100 y para la UltraSparc T=50.

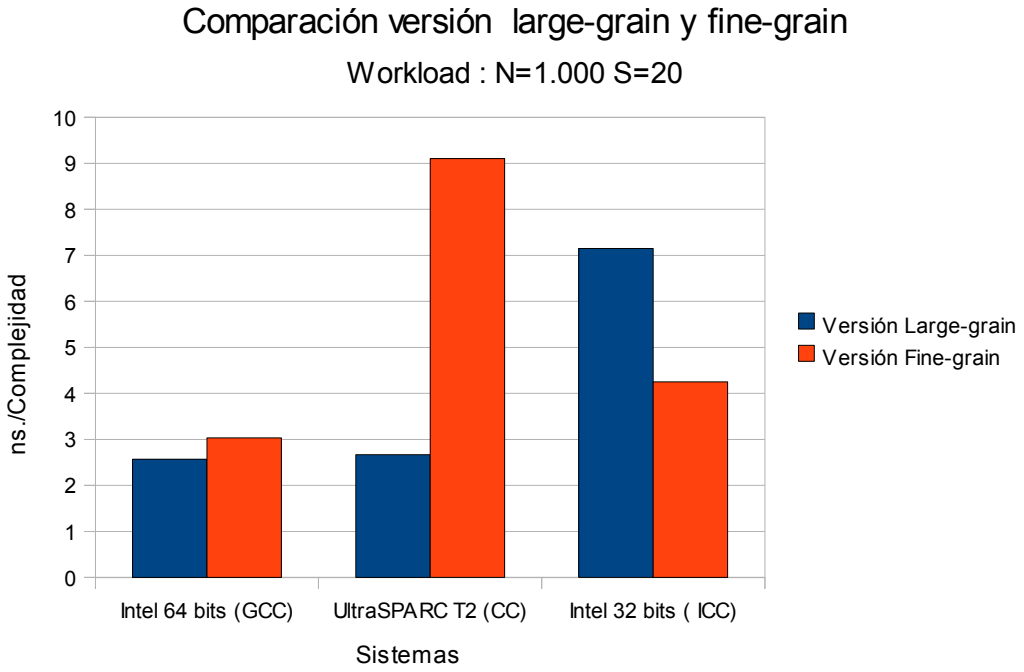


Figura 50: Tiempo de ejecución divididos por complejidad, para la ejecución de las 2 variantes paralelas en los 3 procesadores con el máximo número de threads para ambas variantes y el mejor valor de T para cada procesador en la variante fine-grain.

Con el Workload (N=10.000, S=2) los valores de T usados son para el sistema x86-64 y x86-32 T=500 y para la UltraSparc T=50.
En este Workload tan solo podemos ejecutar la aplicación con 4 threads ya que con 2 secuencias sólo son posibles 4 alineamientos. Si probamos de ejecutar la aplicación aumentando el número de secuencias de manera que cada thread pueda realizar un alineamiento nos encontramos con un problema de memoria, ya que se supera el tamaño de memoria disponible. Por lo tanto podemos concluir que con secuencias muy grandes la versión large-grain no es viable debido al espacio en memoria que ocupará cada una de las matrices con las que trabaja cada uno de los threads (400 MB. aproximadamente cada matriz)

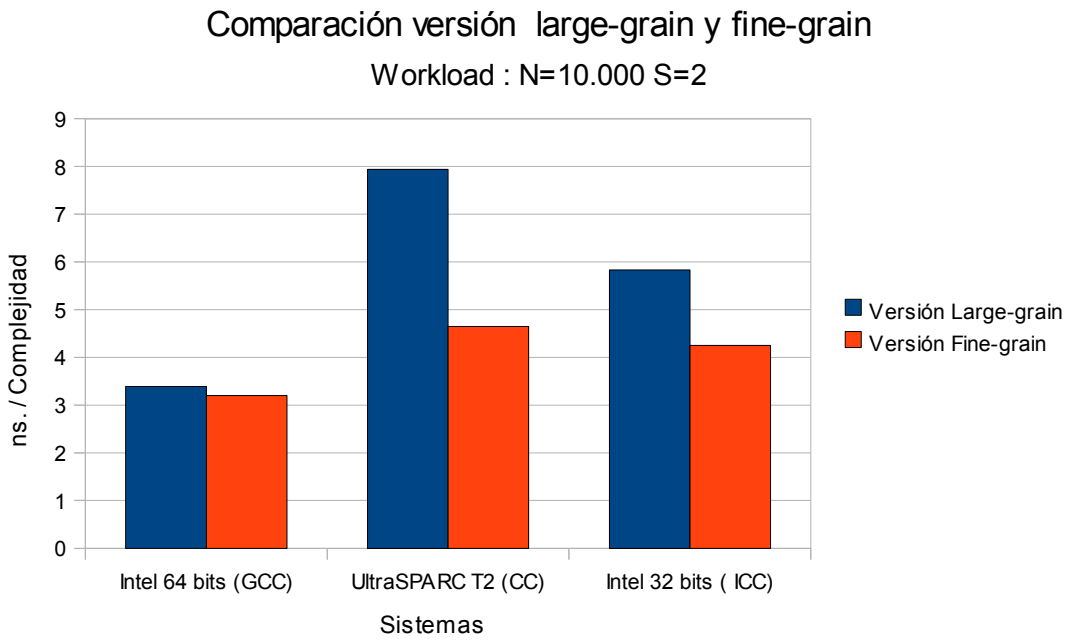


Figura 50: Tiempo de ejecución divididos por complejidad, para la ejecución de las 2 variantes paralelas en los 3 procesadores con el máximo número de threads para ambas variantes y el mejor valor de T para cada procesador en la variante fine-grain.

Se han extraído una serie de conclusiones a raíz de los resultados mostrados :

- Si comparamos las 2 variantes paralelas nos encontramos con que si tenemos pocas secuencias y de tamaño grande la variante de granularidad fina es la que funciona mejor. Por contra, si tenemos muchas secuencias y de tamaño pequeño la variante de granularidad gruesa es la que va mejor.

- En el procesador Intel de 64 bits el multithreading en ambas variantes resulta beneficioso. Aunque el rendimiento se degrada al aumentar el tamaño de las secuencias. Seguramente debido a que para tratar secuencias grandes los datos no caben en la memoria caché del procesador, y hay que obtenerlos de la memoria principal. El hecho de utilizar un método de descomposición como el WaveFront también nos dificulta la tarea de la paralelización ya que al tener que ir calculando cada momento la diagonal hemos de esperar a que todos los threads acaben de calcular su parte para poder pasar a calcular los siguientes elementos (desbalanceo de carga).

- Si ejecutamos la versión de granularidad gruesa y la versión de granularidad fina con 1 thread sobre el procesador UltraSparc T2 los tiempos por iteración son muy altos. Los motivos principales son el alto tiempo de acceso a Memoria Principal y cache L2, el cual no puede compensarse con la ejecución de instrucciones del thread, porque la planificación de instrucciones es en orden.
  El multi-threading en ambas variantes es muy positivo porque resuelve los anteriores problemas. Los tiempos de espera se pueden solapar con la ejecución de instrucciones de otros threads. Por este motivo a medida que dispone de más threads para ejecutar, el tiempo disminuye.

- En el procesador Intel de 32 bits la variante de granularidad gruesa no resulta beneficioso ya que los tiempos obtenidos son bastante peores que la versión en serie.
  Mientras que la variante de granularidad fina el multi-threading es positivo ya que obtenemos un speedup de 1.4 en comparación a la versión serie.

6. Conclusiones y lineas abiertas

El objetivo principal de este trabajo consiste en realizar un proceso de análisis y optimización de un problema. En nuestro caso se trataba de implementar eficientemente un algoritmo para ser ejecutado en procesadores multicore. El primer paso consiste en la implementación del problema en un lenguaje de programación. Seguidamente se planifican y ejecutan los experimentos en procesadores de características diferentes. Finalmente se interpretan los resultados obtenidos para poder extraer conclusiones que permitan optimizar el rendimiento del algoritmo en cada procesador.

Este proceso de análisis, implementación, ejecución, interpretación de resultados y posterior optimización hemos visto que resulta más complicado de lo que parece. Esto es debido a que para poder llevar a cabo la optimización se requiere un conocimiento muy a fondo tanto del comportamiento del procesador cómo del programa. Este conocimiento se va refinando a base de pruebas y medidas, esto es debido a que no podemos ver los problemas internos que provocan la bajada del rendimiento, tan sólo podemos hacer pruebas para obtener datos de forma indirecta, ya sea en forma de tiempo, número de fallos de caché o número de fallos de predicción.
Hemos visto que el proceso de optimización de un problema puede resultar complejo debido a que al mejorar un aspecto de nuestro programa podemos estar empeorando otro. Un ejemplo claro es la relación entre sincronización y balanceo de cómputo que encontramos en la variante de grano fino, en la cual el hecho de mejorar la sincronización hace que empeore el balanceo de cómputo y a la inversa.

A nivel del trabajo, hemos visto que la paralelización del algoritmo en threads ha sido efectiva en casi todos los experimentos realizados. La única excepción ha sido el procesador Intel 32 bits en la variante de grano grueso que apenas ha llegado a funcionar mejor que la versión serie. Las 2 variantes que hemos analizado poseen ventajas e inconvenientes que hacen que nos decantemos por una o por otra dependiendo del tipo y número de secuencias con las que queramos trabajar. Para alinear varias secuencias entre ellas la variante Large-Grained va mejor en cuanto a tiempo de ejecución , en cambio si queremos alinear pocas secuencias y muy grandes la variante Fine-Grained es más rápida.

En cuanto a los procesadores analizados podemos concretar que el procesador Intel de 64 bits ofrece el mejor rendimiento tanto en la versión serie, como en las 2 variantes paralelas. El procesador Intel de 32 bits obtiene un rendimiento 2 veces menor que su homólogo de 64 bits. En cambio el procesador UltraSPARC obtiene un rendimiento 6 y 4 veces menor que utilizando los procesadores Intel de 64 y 32 bits correspondientemente (ejecutando con un thread). En el UltraSPARC T2, es necesario ejecutar las variantes paralelas con 32 threads, para que el rendimiento se aproxime al obtenido en el resto de procesadores (ejecutando en éstos 2 threads). Lo que nos lleva a afirmar que en el UltraSPARC se requiere un esfuerzo especial para obtener (ejecutando una aplicación), un rendimiento cercano al obtenido en los procesadores Intel Dual Core.

6.1. Lineas futuras de investigación

A continuación se presentan líneas futuras de investigación que por falta de tiempo no se han podido investigar con mas detalle.

- Investigar con más profundidad el uso de los recursos de cálculo para descubrir cual es el cuello de botella del rendimiento.

- Análisis de rendimiento la aplicación ejecutada en un procesador con una arquitectura diferente, como pudiera ser AMD

- Estudio en más profundidad de los fallos en caché para poder analizar posibles problemas de rendimiento.

- Explorar la viabilidad de usar instrucciones SSE para realizar la paralelización en los procesadores Intel minimizando lo mas posible la dependencia de datos de la matriz de puntuaciones.

- Uso de la aplicación VTUNE para realizar un análisis del rendimiento del programa.

7. Bibliografía

Referencias :

[1] Edited by David C. Brook (2006). "*Understanding Moore's Law*". Chemical Heritage, Philadelphia.

[2] Jurij Silc, Borut Robic, Theo Ungerer (1999).*"Processor Architecture: From Dataflow to Superscalar and Beyond".* Springer-Verlag, Alemania

[3] Hennessy, J.L. y Patterson, D.A. (1996). "*Computer Architecture: A Quantitative Approach*" Morgan Kaufmann Publishers, San Francisco.

[4] Michael J. Flynn (1995)."*Computer Architecture : Pipelined and parallel processor design".* Jones and Bartleet Publishers, London.

[5] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald y Ramesh MenonParallel (2001). "*Programming in OpenMP*". Academic Press, London.

[6] Chris R. Calladine, Horace D. Drew, Ben F. Luisi y Andrew A. Travers (1992). *"Understanding DNA"*. Elsevier Press, California.

[7] David W. Mount (2004). *"Bioinformatics: sequence and genome analysis. Cold Spring"* . Cold Spring Harbor, New York.

[8] Julian Rubin (2009). "*Bioinformatics Encyclopedia : Needleman-Wunsch Algorithm*". Último Acceso : Mayo 2010. Última actualización : Enero 2010.
Página Web : *"http://www.juliantrubin.com/encyclopedia/bioinformatics/needleman_wunsch_algorithm.html"*

[9] Vipin Kumar, Marina L. Gavrilova, Chih Jeng Kenneth Tan (2003)."*Computational Science and Its Applications – ICCSA 2003*". Pierre L'ecuyer, Montreal

Firma del autor de la memoria:

Firmado por Carlos Pons Noguera
Bellaterra, 22 de Junio de 2010

RESUMEN

Este trabajo analiza el rendimiento del algoritmo de alineamiento de secuencias conocido como Needleman-Wunsch, sobre 3 sistemas de cómputo multiprocesador diferentes. Se analiza y se codifica el algoritmo serie usando el lenguaje de programación C y se plantean una serie de optimizaciones con la finalidad de minimizar el volumen y el tiempo de cómputo. Posteriormente, se realiza un análisis de las prestaciones del programa sobre los diferentes sistemas de cómputo.
En la segunda parte del trabajo, se paraleliza el algoritmo serie y se codifica ayudándonos de OpenMP. El resultado son dos variantes del programa que difieren en la relación entre la cantidad de cómputo y la de comunicación.
En la primera variante, la comunicación entre procesadores es poco frecuente y se realiza tras largos periodos de ejecución (granularidad gruesa). En cambio, en la segunda variante las tareas individuales son relativamente pequeñas en término de tiempo de ejecución y la comunicación entre los procesadores es frecuente (granularidad fina).
Ambas variantes se ejecutan y analizan en arquitecturas multicore que explotan el paralelismo a nivel de thread. Los resultados obtenidos muestran la importancia de entender y saber analizar el efecto del multicore y multithreading en el rendimiento.

- **Palabras clave:** Multicore, multithread, rendimiento, OpenMP, alineamiento, ADN.

RESUM

Aquest treball analitza el rendiment de l'algoritme d'alineament de seqüències conegut com a Needleman-Wunsch sobre 3 sistemes de còmput multiprocessador diferents. S'analitza i es codifica l'algoritme sèrie emprant el llenguatge de programació C i es plantegen una serie d'optimitzacions amb la finalitat de minimitzar el volum i el temps de còmput. Posteriorment es realitza un anàlisis de les prestacions del programa sobre els diferents sistemes de còmput. En la segona part del treball, es paral·lelitza l'algoritme sèrie i es codifica ajudant-nos de OpenMP. El resultat són dues variants del programa que difereixen en la relació entre la quantitat de còmput i la de comunicació.
En la primera variant, la comunicació entre processadors es poc habitual i es realitza després de llargs períodes d'execució (granularitat gruixuda). En canvi, en la segona variant les tasques individuals s'executen relativament rapides i la comunicació entre els processadors es freqüent (granularitat fina).
Ambdues variants s'executen i s'analitzen en arquitectures multicore que exploten el paral·lelisme a nivell de thread. Els resultats obtinguts ens mostren la importància d'entendre i saber analitzar l'efecte del multicore i el multithreading en el rendiment.

- **Paraules clau:** Multicore, multithread, rendiment, OpenMP, alineament, ADN.

ABSTRACT

This research analyzes the performance of three multiprocessor computing nodes solving the seqüence alignment algorithm known as Needleman-Wunsh. First of all, the algorithm is analyzed and coded using the C language. We raise a series of optimizations with a common goal : minimize memory requirements and reduce computation time. Right afterwards we analyze the program's performance over the three computation nodes. In the second part of the research the sequential algorithm is parallelized using OpenMP. Two program variations are designed, these two variations differs between them in the amount of computation and the comunication.
On the first variation the comunication between processors is rarely common and only occurs after long time periods . On the second variation the tasks are processed rapidly and the communication between processors is common.
Both variations have been implemented and executed in multicore architectures that exploits thread-level parallelism. The result shows the importance of understanding and knowing how to analyze the effect of multicore and multithreading performance.

- **Keywords:** Multicore, multithread, performance, OpenMP, alignment, DNA.