
UNIVERSIDAD
DE MÁLAGA

Departamento de Arquitectura de Computadores

TESIS DOCTORAL

# Patrones de Paralelismo de Alto Nivel Basados en Tareas

Antonio J. Dios García Martín

Julio de 2013

Dirigida por:
Rafael Asenjo Plaza,
María Ángeles González Navarro

Dr. D. Rafael Asenjo Plaza.
Profesor Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

Dra. Dña. Mª Ángeles González Navarro.
Profesora Titular del Departamento de Arquitectura de Computadores de la Universidad de Málaga.

**CERTIFICAN:**

Que la memoria titulada "Patrones de Paralelismo de Alto Nivel Basados en Tareas", ha sido realizada por D. Antonio J. Dios García Martín bajo nuestra dirección en el Departamento de Arquitectura de Computadores de la Universidad de Málaga y constituye la Tesis que presenta para optar al grado de Doctor en Ingeniería Informática.

Málaga, Julio de 2013

Dr. D. Rafael Asenjo Plaza.
Codirector de la tesis.

Dra. Dña. Mª Ángeles González Navarro.
Codirectora de la tesis.

A mi mujer y toda mi familia

# Agradecimientos

Esta tesis es el producto de la colaboración directa e indirecta de multitud de personas a las que quisiera dirigir unas breves palabras de agradecimiento con estas líneas.

En primer lugar, me gustaría dar las gracias a mis directores: Rafael Asenjo y María Ángeles Navarro, por estos años de dedicación, por haberme transmitido sus conocimientos, por sus consejos y sobre todo por la gran labor en la dirección de esta tesis para poder avanzar siempre por el camino adecuado. He de agradecerles su ayuda no solo en el ámbito académico, sino también por su esfuerzo al apoyarme en los momentos que la presión de la tesis fue más dura y las fuerzas me flaqueaban. También quiero agradecerle a Francisco Corbera, profesor titular perteneciente al grupo de investigación, por su colaboración en los momentos en los que su ayuda fue necesaria, por sus grandes ideas y por transmitirme su entusiasmo en la investigación. No puedo olvidarme agradecerle a Emilio L. Zapata, por haberme dado la oportunidad de realizar la tesis doctoral en el departamento cuando él era el Director. Por otro lado, quisiera acordarme en estas líneas de David Padua y María Garzarán, profesores de la Universidad de Illinois, con quienes tuve la suerte de trabajar durante mi estancia en la universidad de Illinois (Estados Unidos). Tampoco quiero olvidarme de las personas que he tenido ocasión de conocer en los distintos congresos en los que he participado, en especial de Arturo, cuya ayuda inestimable en el congreso HPCC de Melbourne me sirvió para poder afrontar con mayor seguridad aquella experiencia.

También debo agradecer la financiación recibida de la Consejería de Innovación, Ciencia y Empresa de la Junta de Andalucía, a través de los proyectos de investigación CICYT P08-TIC35008, y al Ministerio de Ciencia e Innovación del Gobierno de España, a través de los proyectos de investigación CICYT TIN2006-01078, CICYT TIN2010-16144 y Consolider-Ingenio2010, CSD2007-00050.

Por último, quisiera agradecer a mis compañeros de departamento, que han contribuido con un gran ambiente a que estos años sean inolvidables, y por supuesto por su inestimable colaboración cuando la he necesitado: Juan Lucena, Alberto Sanz, Pilar Poyato, Ricardo Quislant, Miguel Ángel Mesa, Lídea Fernández, Iván Romero, Antonio

Muñoz, Manuel Cervilla, Manuel Pedrero, Oscar Torreño, Pau Corral, Francisco Jaime, Sergio Muñoz, Sergio Varona, Juan Villalba y Carlos García. En especial a aquellos con los que he tenido la oportunidad de compartir laboratorio con los que sin duda mantendré una relación de amistad. Quiero agradecerle a mis amigos, compañeros de la carrera y del colegio, por haber estado pendientes de mi trayectoria estos años y haberme apoyado y dado ánimos para continuar con esta labor. No puedo olvidar tampoco en estas líneas a mis padres, ya que sus experiencias y conocimientos en el mundo de la investigación universitaria me han servido como soporte y guía para progresar en este difícil objetivo. Por supuesto a mis hermanas, Guille y María, a sus maridos y sobre todo a mis sobrinos, gracias por estar a mi lado y hacerme sentir siempre querido en todos los momentos de mi vida. También me gustaría agradecer al Stmo Cristo Coronado de Espinas de la cofradía de los Estudiantes, como buen Doctor, supo aleccionarme una clase magistral aquella madrugada del Lunes Santo que me permitió encontrar las fuerzas y el camino para poder acabar este proyecto cuando más lo necesitaba. Por otro lado, quisiera dedicarles a mis abuelos este trabajo, se que ellos estarían muy orgullosos de mi. Pero sin lugar a dudas, si a alguien he de agradecerle su apoyo fiel y sincero, si alguien ha soportado mis momentos anímicamente inestables y supo hablarme claro para que reaccionase, ella es a mi mujer, Mar. Porque nuestro noviazgo comenzó con en esta etapa de mi vida que hoy cierro y culminó el pasado mes 20 de octubre con nuestra boda. Has sufrido mis penas, mi cansancio, mis enfados, y con paciencia has sabido animarme, quererme y cuidarme, gracias.

# Resumen

La programación paralela es un requerimiento en la era multicore. Una aproximación prometedora para explotar las arquitecturas paralelas consiste en explotar los distintos patrones de paralelismo. En esta tesis nos centramos en los patrones de paralelismo de tipo “pipeline” y “wavefront”. En cuanto a los paradigmas de programación paralela, hemos encontrado que el modelo basado en tareas se adapta perfectamente a estos dos patrones en arquitecturas de memoria compartida. Entre las razones que justifican este comportamiento cabe destacar que las tareas son más ligeras que los threads, permiten un mayor nivel de abstracción desde el punto de vista del programador lo que repercute en una mayor productividad y que los planificadores de tareas basados en *work-stealing* pueden balancear mejor la carga de trabajo.

Las librerías basadas en tareas, como por ejemplo Intel Threading Building Blocks (TBB), son herramientas que ayudan a los programadores a desarrollar código paralelo de un modo productivo, gracias a constructores de alto nivel como las plantillas. TBB proporciona una plantilla para expresar paralelismo de pipeline a alto nivel pero está limitada a estructuras simples de pipeline y no soporta la posibilidad de que una etapa del pipeline genere varios elementos de salida por cada uno de entrada. En esta tesis, hemos resuelto esa limitación desarrollando un nuevo filtro para este tipo de etapas, que denominaremos etapas “multioutput”. Experimentalmente y mediante un modelo analítico hemos comprobado que nuestro nuevo filtro aventaja a otras alternativas, principalmente gracias a que reduce el overhead relativo al uso de memoria y a la gestión de tareas al tiempo que promueve el paralelismo de grano fino lo que mejora la utilización de los recursos.

En cuanto al patrón de tipo wavefront, ninguna librería basada en tareas proporciona soporte de alto nivel para una implementación productiva. Para cubrir esta necesidad, como primer paso, hemos comparado distintas implementaciones de bajo nivel de este tipo de problemas usando distintas librerías que representan el estado del arte en el modelo basado en tareas: TBB, OpenMP 3.0, Cilk y CnC. Discutimos la diferencias y particularidades de cada implementación desde el punto de vista del programador.

Además realizamos distintos experimentos para identificar los factores que pueden limitar el rendimiento en cada una de las implementaciones. Encontramos que TBB provee una serie de características que permiten una implementación más eficiente y que en estos problemas, si el usuario guía al planificador, puede obtener un mejor rendimiento de la memoria cache aprovechando la característica de "paso de tareas" de TBB. Sin embargo, TBB no posee ninguna plantilla para implementar problemas de tipo wavefront por lo que hemos diseñado una al efecto. Para usar la nueva plantilla, el programador sólo debe proporcionar un fichero de definición indicando el patrón de dependencias y otro con el código que se ejecuta para cada celda del grid. Posteriormente, hemos incorporado varias optimizaciones a esta plantilla, de entre las cuales destacamos la gestión automática del tiling.

# Índice de Contenido

# Índice de Figuras

# Prefacio

Las arquitecturas de los computadores modernos son cada día más complejas. Ya encontramos procesadores con 4, 8 o más cores incluso en los teléfonos móviles y tabletas. Por otro lado, el usuario final requiere que las aplicaciones se ejecuten de manera rápida y eficiente en estas arquitecturas multicore, de forma que se aprovechen de forma razonable todos los recursos hardware. Para ello podemos aplicar técnicas de programación o computación paralela: aquella en la que se ejecutan varias instrucciones o procesos de manera simultánea en distintas unidades de computación. En las últimas décadas, la comunidad científica ha realizado grandes esfuerzos para paralelizar aplicaciones que tienen un alto coste computacional. Dentro de la variedad de problemas que admiten eficientes soluciones mediante programación paralela, nos centraremos en los de tipo *stream*, en los que hay que procesar los elementos de un flujo de datos. Este tipo de problemas aparecen tanto en aplicaciones multimedia como de índole científica, y se adaptan bien al modelo de paralelismo "Simple Instrucción Múltiple Dato" (SIMD). Por otro lado, la programación paralela presenta dificultades añadidas a la programación clásica ya que aparecen nuevos patrones y conceptos que hay que tener en cuenta, como sincronización, creación dinámica de tareas, estructuras de datos concurrentes, etc. Además, en muchas ocasiones las aplicaciones han de ser desarrolladas por personas no expertas en programación paralela. Es por tanto también importante que se proporcionen herramientas que faciliten la implementación paralela de estos problemas.

De entre todos los modelos y paradigmas de programación paralela que han surgido en los últimos años, en esta tesis nos centramos en el paradigma basado en tareas. De entre ellos, dedicaremos mayor atención a TBB (Threading Building Blocks), ya que hemos comprobado que ofrece funcionalidades de bajo nivel que ayudan a optimizar la ejecución, lo que redunda en un mejor rendimiento que el de los otros modelos basados en tareas. Complementáriamente, para facilitar el uso de las eficientes primitivas de bajo nivel, TBB proporciona varias plantillas de alto nivel para implementar patrones de diseño habituales en programación paralela, lo que redunda en una mejora de la productividad para el programador no experto.

Dentro de esta corriente, nuestro trabajo persigue los siguientes objetivos:

- Validar el modelo de programación basado en tareas para el patrón pipeline (un subconjunto de problemas de tipo *stream*), utilizando una plantilla que TBB ofrece para dicho patrón. Para ello se caracterizan problemas pipeline del conjunto de Benchmarks PARSEC, y se estudia la implementación de estos problemas en el modelo basado en tareas.
- Implementar optimizaciones en la plantilla pipeline de TBB para casos específicos que no son contemplados en la versión original de la plantilla.
- Validar el patrón de diseño wavefront (otro subconjunto de problemas de tipo *stream*) y caracterizar su comportamiento bajo el modelo basado en tareas. Destacar cuál de los frameworks basados en tareas que representan el estado del arte es el más apropiado para implementar aplicaciones de tipo wavefront.
- Diseñar e implementar una plantilla de alto nivel que facilite programar problemas de tipo wavefront de manera sencilla pero eficiente.

Para la consecución de estos objetivos, en primer lugar implementamos con TBB dos problemas de tipo pipeline pertenecientes al conjunto de Benchmarks PARSEC y evaluamos su rendimiento. Tras comprobar que la plantilla de TBB no cubre las necesidades de algunos problema de tipo pipeline, proponemos una plantilla mejorada y la comparamos, tanto experimental como analíticamente, con otras implementaciones alternativas.

En cuanto al patrón wavefront, primero comparamos varias implementaciones de un problema wavefront utilizando TBB, OpenMP, Cilk y CnC. Tras comparar los resultados obtenidos y destacar las ventajas e inconvenientes de cada uno, se concluye que TBB ofrece funcionalidades que pueden ser explotadas para conseguir implementaciones más eficientes de problemas de tipo wavefront. Con objeto de proporcionar la misma productividad disponible en TBB para problemas de tipo pipeline, decidimos diseñar una plantilla que también facilite a los programadores desarrollar aplicaciones de tipo wavefront. Esta plantilla se valida aplicándola a varios problemas reales y evaluando su rendimiento. También utilizamos herramientas de *profiling* para caracterizar el overhead de la nueva plantilla, detectándose las principales limitaciones y aspectos a mejorar, sobre todo en problemas de grano fino. Esto da pie a la implementación de las optimizaciones que permiten superar las limitaciones encontradas.

En esta memoria se recogen las reflexiones, estudios, detalles de implementación y resultados experimentales sobre los aspectos discutidos en los párrafos anteriores. La organización de esta memoria se describe a continuación:

En el capítulo 1 definimos dos casos de problemas de tipo *stream*, en concreto los problema de tipo pipeline y de tipo wavefront. También comparamos el modelo basado en tareas y el modelo basado en threads, explicando detalladamente por qué el modelo basado en tareas encaja perfectamente en este tipo de problemas, así como cuáles son los frameworks que representan el estado del arte en este modelo.

A continuación, el capítulo 2 analiza los programas ferret y dedup, representativos del patrón pipeline y pertenecientes al conjunto de Benchmarks PARSEC. Se estudia el rendimiento ofrecido por distintas implementaciones basadas en la plantilla para el patrón pipeline de Intel Threading Building Blocks. También proponemos una optimización para esta plantilla, que la dota de mayor flexibilidad y aplicabilidad, junto con un modelo analítico que permite comparar nuestra propuesta con otras alternativas.

En el capítulo 3 cambiamos el foco de atención y pasamos del patrón pipeline al de wavefront. Realizamos un análisis del rendimiento de códigos wavefront usando distintos frameworks de programación basados en tareas (TBB, Cilk, OpenMP y CnC). La contribución más destacable de este capítulo se encuentra en el apartado 3.3, donde presentamos una plantilla basada en TBB para programar de manera sencilla y productiva problemas de tipo wavefront.

Seguidamente, en el capítulo 4 proponemos varias mejoras en la implementación de nuestra plantilla para códigos wavefront. Estas mejoras permiten aumentar la eficiencia de los ejecutables generados reduciendo el overhead. De entre estas mejoras destacamos la que implementa el tiling automático que permite cómodamente obtener aceleraciones aceptables incluso para problemas de grano muy fino. Estas mejoras no afectan a la facilidad de uso de la plantilla, ya que por ejemplo, el tamaño de tile se computa automáticamente.

Para acabar, en el capítulo 5 sintetizamos las principales aportaciones de esta tesis y discutimos posibles líneas de trabajo futuro.

# 1 Introducción a los problemas de tipo stream

El avance tecnológico está multiplicando el número de cores en los procesadores actuales. Esto está haciendo que cada vez cobre más relevancia la programación paralela. Una de las grandes dificultades de la programación paralela está en la sincronización de las distintas partes de un programa y en el acceso a los recursos compartidos. Por tanto, para paralelizar un código es muy importante tener experiencia en programación concurrente. En caso contrario, a un programador inexperto puede resultarle complicado obtener al mismo tiempo un buen rendimiento y un correcto funcionamiento de su programa.

Dado que los procesadores multicore cada vez están más omnipresentes, la programación paralela es clave para garantizar el éxito de las máquinas multicores. En este sentido, la investigación está orientándose en dos direcciones principales: i) desarrollo de lenguajes para programación paralela, como puede ser X10 [84], Chapel [14] y Fortress [29], o extensiones para programación paralela para lenguajes ya existentes, como UPC[18], Co-Array Fortran [58], Titanium[39]; y ii) desarrollo de librerías paralelas altamente optimizadas que encapsulan paralelismo de datos [31, 62] o paralelismo de tareas [71]. Ambos casos tienen como objetivo liberar al programador de tener que lidiar con la complejidad de la arquitectura, proveyendo altos niveles de abstracción que permiten expresar el paralelismo de una aplicación y confiar en que el sistema explote eficientemente la concurrencia.

Los problemas que admiten una implementación paralela son muy numerosos, pero en los últimos tiempos se está realizando un gran esfuerzo para conseguir mayor eficiencia en los problemas de *streaming*. Estos problemas se encuentran en algoritmos de procesamiento de imágenes o en problemas de índole científica. Son problemas basados en stream de datos (audio, vídeo, DSP, encriptado, etc, en los que aparece una serie de

computaciones regulares y repetidas que se pueden expresar como etapas (tareas) independientes que se comunican o sincronizan. En concreto nos centraremos en dos clases de problemas relacionados con el *streaming* de datos: los problemas de tipo *pipeline* y los problemas de tipo *stencil*.

En este capítulo explicamos los problemas de tipo *stream*, sus características y las dificultades para implementarlos. Así mismo, discutimos qué modelo de programación paralela (threads vs. tareas) es el más adecuado para abordar estos problemas.

## 1.1. Problemas de tipo *stream*

Los problemas de tipo *stream* son aquellos en los que a cada elemento de una secuencia (*stream*) de datos se le aplica una serie de operaciones o funciones (kernels) que se comunican o sincronizan. Las aplicaciones de tipo *stream* más típicas son las de procesamiento de imagen, vídeo y señales digitales. Se adaptan bien al procesamiento paralelo si estas aplicaciones reúnen las siguientes características:

- Intensidad aritmética elevada. Efectivamente, el número de operaciones aritméticas en relación al número de referencias a memoria o de operaciones de Entrada/-Salida suele ser alto para los problemas de tipo *stream*. En muchas aplicaciones de procesado de señal esta relación puede ser superior a 50:1, pero en este trabajo también intentaremos conseguir eficiencias razonables en problemas que exhiben una intensidad aritmética menos favorable.

- Paralelismo de datos. Este tipo de paralelismo existe cuando se aplica una misma función a todas las entradas de un flujo (*stream*) de datos. Cuando no existen dependencias entre dichos datos, la función se puede aplicar en paralelo a distintos datos siguiendo un modelo SIMD (Single Instruction Multiple Data).

- Localidad de los datos. Es habitual en el procesamiento de señal o de datos multimedia que los datos se generen una vez, a continuación se lean una o dos veces y luego no se necesiten nunca más. Por tanto la localidad temporal no suele destacar y de hecho las arquitecturas específicas para procesado en stream, como DSPs o GPUs, no incluyen cachés, o si lo hacen son de pequeño tamaño. Sin embargo, cuando los datos se leen de memoria, por ejemplo en las aplicaciones tipo *stencil* o *wavefront*, sí existe bastante localidad espacial ya que los datos que alimentan el stream en un momento dado suelen estar en posiciones cercanas en el espacio de datos.

En este trabajo vamos a abarcar dos clases de problemas relacionados con el *streaming*. En primer lugar hablaremos de los problemas de tipo *pipeline* en los que un flujo de datos recorre las distintas etapas (filtros) del pipeline y en cada etapa a los datos se les aplica una función o kernel independiente. En este caso, entre cada etapa y la siguiente se produce una comunicación de datos siguiendo un esquema productor-consumidor. En segundo lugar tenemos las aplicaciones de tipo *stencil* en las que se procesaran elementos de un grid de datos almacenados en memoria y se requiere un flujo de esos datos desde memoria al procesador. Sobre ese flujo se aplica la misma función o kernel.

### 1.1.1. Pipeline

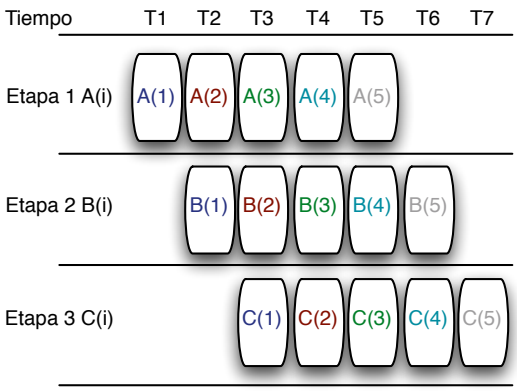
El patrón *pipeline* permite solapar en el tiempo distintas iteraciones de un bucle, de manera que puedan ejecutarse en paralelo aunque haya dependencias entre ellas [53]. Consideremos el ejemplo de la figura 1.1.

```
1 for (int i=1; i<N; i++){
2   A(i);
3   B(i);
4   C(i);
5 }
```

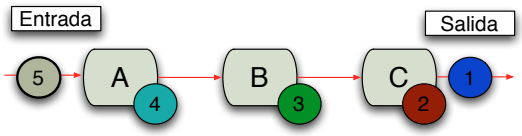
Figura 1.1: Bucle a ejecutar siguiendo una estrategia de tipo *pipeline*.

En el bucle tenemos A(i), B(i) y C(i) que son distintas funciones aplicadas a cada dato $i$. Supondremos que el recurso que ejecuta cada función, A, B y C está ocupado durante la computación de cada dato. Es decir, no podemos solapar la ejecución de A(i) con la de A(j), y de igual forma para B y C. También supondremos que C(i) depende de B(i), que a su vez depende de A(i), por lo que el cuerpo del bucle `for` se debe ejecutar en orden. Es decir, en una ejecución serie, la secuencia de operaciones sería: A(1), B(1), C(1), A(2), B(2), C(2), A(3), B(3), C(3), etc. Sin embargo, si no existen dependencias entre distintas iteraciones, el bucle es paralelo y podríamos ejecutar A(i+1) en paralelo con B(i) o C(i). Esta planificación de la ejecución de las distintas iteraciones de A, B y C se muestra en la figura 1.2(a), donde podemos ver como en una ejecución *pipeline* podemos solapar la ejecución de A(2) con B(1), o de A(3), B(2) y C(1), etc. Claramente, el pipeline tiene una latencia inicial de llenado y otra final de vaciado en las que algunos recursos (o etapas del pipeline), pueden quedar ociosos. Por ejemplo, si nuestro bucle tiene 5 iteraciones, en la figura 1.2(a) vemos como durante el período de tiempo T1, B y C están ociosos, en T2, sólo C está inactivo, pero a partir de T3 y hasta T5 todas las etapas están ocupadas. A partir de T6 se produce el vaciado del pipeline y de nuevo

aparecen recursos infrautilizados. En la figura 1.2(b) se muestra un diagrama con las tres etapas del pipeline y cómo estas van procesando distintos elementos del flujo de datos.




(a) Procesamiento de cada etapa de un pipeline en el tiempo.




(b) Flujo de datos procesados en pipeline

Figura 1.2: Ejemplo de procesamiento de una secuencia de datos en pipeline.

Existen dos tipos de *pipeline*. Los *pipeline* lineales y los no lineales. Un *pipeline* lineal consiste en una serie de etapas de procesamiento o filtros que definen un flujo de control lineal que se aplica sobre un *stream* de datos. Un *pipeline* no lineal puede estar configurado para que los datos puedan seguir distinto flujos de control. En un *pipeline* no lineal también puede haber realimentación de etapas, es decir, puede haber ciclos.

Cuando la duración de la ejecución de todas las etapas es similar, el patrón es más eficiente ya que el rendimiento del *pipeline* viene determinado por la etapa más lenta. Por otro lado, cuando existe una etapa más rápida que la siguiente es necesario la introducción de un buffer a la entrada de la etapa más lenta para poder almacenar temporalmente el stream de datos de entrada [63]. Además, en algunas ocasiones una etapa puede dividir los datos en sub-elementos que son transferidos a través del *pipeline* por lo que es necesario agruparlos nuevamente en otra etapa. Algunas veces es necesario reordenar los elementos en los *pipeline* no lineales ya que estos pueden llegar desordenados.

Para medir el rendimiento de un *pipeline* utilizamos el *throughput* y la latencia. El *throughput* se define como el número de elementos que se procesan por unidad de tiempo una vez que el *pipeline* está lleno. Por ejemplo, si la salida de un *pipeline* es una secuencia de imágenes renderizadas para ser vistas como una animación, entonces el *pipeline* debe tener suficiente *throughput* para generar las imágenes necesarias para conseguir la ratio de *frames* por segundo adecuada. La latencia es el tiempo que tarda un elemento desde que entra en el *pipeline* hasta que sale del cauce. En algunas ocasiones, se debe llegar a un compromiso entre el *throughput* y la latencia. Introducir más etapas puede reducir el *throughput* porque podemos tener más elementos procesándose en paralelo. Sin embargo, debido al overhead que se introduce por la comunicación entre las distintas etapas, un elemento tardará más en ser procesado completamente por el *pipeline* lo que resulta en un aumento de la latencia.

Utilizando el patrón *pipeline* podemos expresar muchas aplicaciones de *stream* en el dominio del procesamiento de la señal digital, gráficos y encriptación [69]. Encontramos aplicaciones de tipo *stream* en el conjunto de *Benchmarks* PARSEC (*The Princeton Application Repository for Shared-Memory Computers*). PARSEC se compone de programas multithread para poner a prueba máquinas multicore, prestando mayor atención a cargas de trabajo asociadas a problemas emergentes de la próxima generación de programas para multiprocesadores de memoria compartida. Los *Benchmarks* están implementados en Pthreads (POSIX Threads [7]) y OpenMP. Dentro del conjunto de aplicaciones existen dos aplicaciones consideradas de *pipeline*: un código de búsqueda de similitud de imágenes (ferret) y otro de compresión (dedup). Examinando el grafo de dependencias para el código de ferret, encontramos que no hay dependencias de datos entre las distintas imágenes a procesar. Para el código dedup, sí existen dependencias entre los items que recorren el pipeline, debido a accesos de escritura/lectura a una tabla hash compartida, dependencias que se resuelven mediante cerrojos (locks). Estas dos aplicaciones están implementadas con librerías de bajo nivel que trabajan con threads, pero en esta tesis propondremos alternativas para implementarlas mediante librerías basadas en tareas y con un interfaz de mayor nivel de abstracción.

## 1.2. Problemas de tipo stencil

Los problemas de tipo *stencil* aparecen en estructuras de mallas o *grids*, donde el cálculo o cómputo de cada elemento se realiza utilizando valores de elementos vecinos [24]. Un *grid* es un caso particular de grafo, en el cual todos los vértices interiores de la malla tienen el mismo número de vecinos. Los problemas de tipo *stencil* suelen estar caracterizados por una regularidad en el patrón de accesos a datos. Esto permite realizar *streaming* de datos desde la memoria principal a las unidades de cálculo aplicando

generalmente la misma función o kernel.

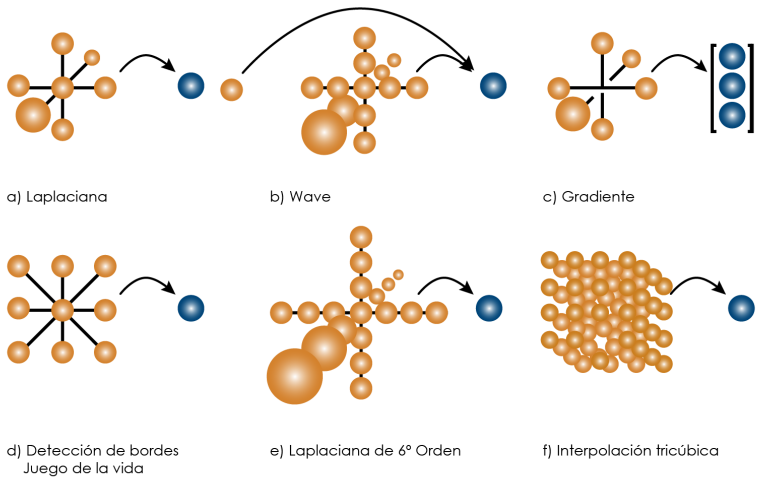



Figura 1.3: Ejemplos de patrones de acceso en problemas de tipo stencil.

Como se ha comentado anteriormente, la intensidad aritmética está definida como el número de operaciones aritméticas dividido por el número de datos (bytes) que se transfieren a la unidad de cálculo y es una medida frecuentemente utilizada para analizar y predecir el rendimiento de un algoritmo *stencil*. Normalmente los problemas *stencil* tiene una intensidad aritmética baja, por lo que el rendimiento de la computación de un problema *stencil* típico está limitado por el ancho de banda con memoria. Habitualmente, el número de operaciones punto flotante por elemento del *grid* es constante y usualmente bajo, comparado con el número de referencias a memoria. Por tanto, la computación *stencil* tiene una intensidad aritmética constante independientemente del tamaño del problema.

Algunas de las aplicaciones de tipo *stencil* son: las ecuaciones diferenciales, métodos de refinamiento de mallas y aplicaciones multimedia como filtros de procesamiento de imágenes (detección de bordes, refinamiento [38], desenfoque [12]). En la figura 1.3 vemos algunos ejemplos de patrones de acceso 2D o 3D típicos en estas aplicaciones *stencil*, en las cuales los datos siempre están mapeados sobre *grid* lineales.

Un código de ejemplo de una computación básica de problema *stencil* se muestra en la figura 1.4. El código *stencil* se corresponde con la estructura Laplaciana de la figura 1.3(a). Como se puede apreciar en el código, para calcular cada elemento en cada

iteración se necesitan los valores de sus vecinos, siendo la función evaluada para todos los puntos del *grid*.

```
for (int x=0; x<X; x++)
 for (int y=0; y<Y; y++)
  for (int z=0; z<Z; z++)
   A[x][y][z]=alfa*u[x][y][z]+
              beta*(u[x-1][y][z]+u[x+1][y][z]+
                    u[x][y-1][z]+u[x][y+1][z]+
                    u[x][y][z-1] u[x][y][z+1]);
```

Figura 1.4: Ejemplo de código stencil que sigue un patrón de tipo Laplaciana.

En aplicaciones de procesamiento de imágenes, la computación *stencil* ocurre en funciones basadas en convoluciones de matrices, las cuales asignan a cada pixel en la imagen resultante el peso de la suma de los píxeles situados alrededor del pixel correspondiente con la imagen de entrada. Estas aplicaciones se corresponden con el ejemplo de la estructura de un *grid* 2D con 8 vecinos que vemos en la figura 1.3(d). Precisamente, otra aplicación que casualmente presenta la misma estructura es el "Juego de la vida" (autómata celular). Conceptualmente, un autómata celular es un *grid* infinito discreto de celdas, las cuales toman un estado finito dentro de una línea de tiempo discreta. En cada etapa se le aplica a cada celda un conjunto de reglas dependiendo de su estado y del estado de sus vecinos. De este modo, el estado global del autómata celular está definido en función del tiempo y una configuración inicial. Un ejemplo de autómata celular fue popularizado por *Gardner* quién describió el juego de la vida [72].

Llamamos "*barrido stencil*" a la aplicación de una computación *stencil* a todos los puntos interiores en un *grid*. Si un barrido *stencil* sólo realiza lecturas, el orden en el que procesemos los elementos es irrelevante y se dice que es de tipo Jacobi [65][15]. Sin embargo, si mientras hacemos el barrido del *grid*, un elemento modificado es también utilizado como dato de entrada para calcular otro elemento, entonces claramente el orden sí es relevante. El método indicará cual es este orden a seguir y se clasifica como de tipo Gauss-Seidel [65][15].

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    B[i][j]=0.25*(A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1]);
```

Figura 1.5: Ejemplo de algoritmo de tipo Jacobi.

El código de la figura 1.5 representa un problema *stencil* de tipo Jacobi. El código

describe un bucle principal donde en cada iteración el valor en un punto es reemplazado por la media de sus vecinos. Este código presenta una estructura simple, cuya paralelización es fácil. La matriz se distribuye en distintos procesadores y cada uno actualiza el subconjunto de elementos de la matriz que se le ha asignado.

```
1 for (x=0; x<X; x++)
2   for (y=0; y<Y; y++)
3     u[x][y]=alfa*u[x][y]+
4              beta*(u[x-1][y]+u[x+1][y]+
5                    u[x][y-1]+u[x][y+1]);
```

Figura 1.6: Ejemplo de algoritmo de tipo Gauss-Seidel.

Por otro lado, en el código de la figura 1.6 podemos ver un código de tipo Gauss-Seidel en el que para computar cada punto del *grid* son necesarios algunos valores de sus vecinos que ya han sido calculados previamente (ahora, la matriz en la que se escribe es la misma que tiene los datos leídos, `u`). Por tanto, en este caso la paralelización del problema no es trivial ya que no basta con calcular cada elemento de la matriz de forma paralela sino que hay que realizarlo de manera ordenada para garantizar que la semántica del código secuencial se respeta, lo que requiere cierta sincronización del flujo de datos.

### 1.2.1. Problemas de tipo wavefront

Dentro del conjunto de problemas de tipo Gauss-Seidel existe un caso particular que aparece en distintas aplicaciones del ámbito científico y multimedia, y que puede ser paralelizado obteniendo un alto rendimiento. Estos problemas, denominados de tipo *wavefront*, aparecen en importantes aplicaciones científicas como aquellas basadas en programación dinámica [81] (algoritmo de Floyd [64], Checkerboard, etc.) o alineamiento de secuencias [4] (secuencias biológicas, etc). Volviendo al conjunto de Benchmarks PARSEC, entre los códigos que se incluyen en dicha suite encontramos la aplicación X264 que implementa un codificador/decodificador de vídeo y que también presenta este patrón.

En el paradigma *wavefront* los datos están distribuidos en un grid multidimensional (de una o más dimensiones) representando un espacio lógico. Por ejemplo, en un problema típico de 2 dimensiones, como el de la figura 1.7, si respetamos las dependencias existentes entre los elementos, observamos un *barrido* diagonal que recorre la matriz desde la esquina superior izquierda hasta la contraria a modo de un frente de onda (ver Fig. 1.7). Esta diagonal (o antidiagonal más propiamente dicho) representa el número de elementos que pueden ser ejecutados en paralelo. El número de elementos independien-

tes crecerá gradualmente hasta la amplitud máxima de la anti-diagonal y a partir de ese instante comenzará a disminuir en el recorrido hacia la esquina inferior derecha de la matriz. Este frente de onda es el que le da el nombre *wavefront* a este tipo de problemas.

```
1 for (i=1; i<n; i++)
2    for (j=1; j<n; j++)
3      A[i,j] = foo(A[i,j], A[i-1,j], A[i,j-1]);
```

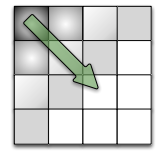


Figura 1.7: Ejemplo de código al que aplicar una parelización de tipo wavefront (izq.) y diagrama del frente de onda correspondiente para el caso de una matriz bidimensional (der.).

## 1.3. Threads vs. Tasks

El objetivo inicial de este trabajo es facilitar la implementación de aplicaciones de tipo streaming en programación paralela, en concreto aquellos que se ajustan al patrón de diseño *pipeline* y al patrón de diseño *wavefront*. Para ello, el primer paso a seguir es verificar qué modelo de programación es más adecuado para estos problemas. En nuestra opinión el modelo de programación basado en tareas se ajusta mejor que el modelo basado en threads en problemas de tipo *pipeline* y en problemas de tipo *wavefront*. A continuación enumeramos las principales razones.

En primer lugar, programar directamente con threads suele ser una elección poco eficiente. Los threads creados por esta librería son threads lógicos, los cuales son mapeados por el Sistema Operativo a threads físicos del hardware. Si creamos pocos threads lógicos (también llamado *undersubscription*, menos threads lógicos que físicos) desaprovechamos los recursos hardware. Si creamos más threads lógicos que físicos (*oversubscription*), el manejo de los mismos incurre en un overhead adicional debido a los costes del cambio de contexto, "cache cooling", así como los derivados de efectos como "lock preemption" y "convoying"[43]. Por tanto, gestionando de manera directa los threads, el programador es el máximo responsable de asignar y aprovechar los recursos hardware disponibles al implementar un programa. Una alternativa para evitar este problema es crear un "*pool*" de threads de modo que una aplicación es la encargada de gestionar y planificar el uso de los threads. De este modo, además, no es necesario crear y destruir continuamente los threads, pues este *pool* los sirve en el momento que es necesario evitando el sobre coste ocasionado por la creación y destrucción de los mismos. Algunos modelos de programación en paralelo como por ejemplo Intel TBB u OpenMP,

facilitan a los desarrolladores los beneficios del tener un *pool* de threads sin tener que estar pendientes de la gestión de los mismos. Usando estos modelos, el programador expresa de manera lógica el paralelismo en sus aplicaciones con tareas. Por tanto, utilizando tareas, el programador puede centrar la atención en expresar los flujos de datos o control del algoritmo, sin preocuparse de las especificación de la planificación del trabajo. En segundo lugar, las tareas son mucho más ligeras que los threads. La creación y destrucción son menos pesadas y es por ello por lo que podemos expresar un paralelismo con una granuralidad mucho más fina. La principal ventaja del uso de tareas respecto a utilizar threads es la ganancia en simplicidad de implementación y transparencia [83] (no existe un planificador oculto que pueda producir un comportamiento inesperado o un rendimiento impredecible).

La principal diferencia entre tareas y threads es que los threads son capaces de recordar un estado después de haber sido suspendidos mientras que las tareas no lo hacen. Esto demanda necesariamente la creación de un espacio privado para cada thread [83]. Por otro lado, las tareas se ejecutan totalmente hasta su finalización y no poseen ningún estado una vez acabadas. Según este modelo, podemos decir que un thread está asociado a un conjunto de tareas. El estado de los threads es mantenido en variables globales. Esto que debería ser una excepción, en la práctica es frecuente en el diseño. El mecanismo de las tareas, es relativamente simple y en la práctica es más rápido. Resumiendo [71]:

1. Sabemos que las tareas son mucho más ligeras que los threads lógicos. De hecho, la creación y sincronización de tareas son uno o dos órdenes de magnitud inferior que los de threads a nivel de sistema operativo.

2. El planificador de threads es parte del Sistema Operativo, no se puede modificar sin recompilar el kernel del mismo, y suele distribuir el trabajo utilizando el algoritmo *round robbin* ya que de esta forma se reparte el tiempo de CPU de forma justa entre los threads. Sin embargo, el planificador de tareas se ejecuta en modo usuario por cada uno de los threads, puede ser modificado o guiado por el programador y sacrifica justicia por eficiencia. Por defecto, los modelos basados en tareas controlan automáticamente el número de threads lógicos creados para evitar situaciones de oversubscription y undersubscription ya que la primera incurre en un overhead extra y la segunda en un bajo aprovechamiento de los recursos.

Existen varios lenguajes o librerías que implementa el modelo de programación basados en tareas: Intel Threading Building Blocks (TBB), OpenMP, Cilk ++, CnC, java.util.concurrent de Java, Microsoft TPL (*Task Parallel Library*) incluida en .Net 4.0, Nanox RT (BSC), Qthreads, así como en lenguajes paralelos como Chapel o X10. Alguno de estos lenguajes o librerías (TBB, Cilk, CnC) tienen un planificador de tareas del tipo *work-stealing*. Este planificador mantiene un *pool* de threads nativos y un con-

junto de tareas por thread preparadas para ser ejecutadas. En la inicialización, se crean los threads necesarios en el *pool* (por defecto un thread sofware por cada core). En este sistema cada thread mantiene su propia cola local conteniendo las tareas que están preparadas para ser ejecutadas. Utilizar colas locales permite evitar problemas de contención derivados del uso de una cola global a la que todos los cores accederían. Los threads ejecutan las tareas de su cola en orden LIFO (la última tarea en entrar será la primera en salir) para un mejor aprovechamiento de la localidad temporal. Sin embargo, si un thread se queda sin trabajo robará de otra cola que no sea la suya en orden FIFO (la primera que entró será la primera en salir) consiguiendo también un mejor uso de la caché ya que robamos a otro thread la tarea "más fría" en su caché. El planificador realiza, de forma transparente al usuario, toda la gestión de tareas, de manera que se consigue un balanceo de carga que sumado a la ligereza de las tareas proporciona una manera eficiente de implementar programas en paralelo.

A continuación dedicamos una subsección a cada uno de los frameworks basados en tareas que se han usado en este trabajo.

### 1.3.1. Intel Threading Building Blocks (TBB)

Intel TBB es una librería de plantillas C++ que está diseñada para asistir a los desarrolladores a la hora de programar en entornos paralelos. La librería de TBB proporciona plantillas para los algoritmos genéricos y contenedores concurrentes más comunes, permitiendo al usuario escribir programas paralelos sin gestionar ni crear threads. La librería está preparada para obtener un rendimiento eficiente y soportar un grano fino de paralelismo a través de tareas. Las tareas son objetos a nivel de usuario que se planifican a través del planificador de tareas de TBB que implementa el modelo *work-stealing*.

TBB posee los siguientes beneficios:

1. Permite especificar tareas en lugar de threads. La mayoría de los paquetes de threads requieren que el usuario cree y gestione los threads directamente. Esta es una labor tediosa porque los threads se gestionan a bajo nivel. Al programar threads directamente, el usuario debe mapear eficientemente las tareas lógicas en threads. Sin embargo, si evitamos el uso de threads nativos se logra una mejor portabilidad, un código más legible, un mejor rendimiento y mayor escalabilidad en general.

2. TBB es compatible con otros lenguajes. Por tanto, podemos añadir código en TBB a cualquier programa escrito con otro lenguaje o librería, como por ejemplo OpenMP o Pthreads, sin tener que reescribir o portar la totalidad del código original.

3. Threading Building Blocks se basa en el concepto de programación genérica. TBB está basado en clases y utiliza "templates" que permiten usar esas clases con distintos tipos de datos. Un buen ejemplo de programación genérica es STL (Standard Template Library) de C++ en la cual las interfaces son especificadas independientemente del tipo de los datos. Por ejemplo una lista puede ser especificada para cualquier tipo (enteros, cadenas, etc.) y con un simple cambio en la declaración instanciar una lista para un tipo particular.

La clase `task` es la primitiva básica de TBB. Sobre esta clase están construidas todas las plantillas de alto nivel. TBB permite con esta clase trabajar directamente con las tareas para proporcionar un control de bajo nivel de la ejecución del trabajo. Cada tarea tendrá un padre y la opción de crear sus propias tareas hijas. Cada vez que una tarea crea otra, ésta pasará a formar parte de la lista de hijos de la tarea salvo que se indique otra cosa. Para crear una nueva tarea existen distintos mecanismo de control. La manera directa es crear las tareas hijas y esperar a que finalice la ejecución de éstas para terminar la tarea padre. También podemos crear las tareas hijas y posteriormente indicar qué trabajo se realizará una vez finalicen las tareas hijas sin bloquear la tarea padre. Sin embargo, aunque esta opción es bastante eficiente, es más difícil de programar que la manera directa.

Para poder utilizar tareas, el usuario debe crear una clase derivada de `TBB::Task` y sobrescribir el método `Task::execute` donde se especifica el trabajo de la tarea. Cuando una tarea crea una instancia de una clase que ha heredado de `TBB::Task` y realiza un `spawn`, la nueva tarea entra en la cola de tareas del thread para ser ejecutada. Este método no es bloqueante y la tarea podrá seguir ejecutándose sin problemas. Sin embargo, es posible utilizar el método `spawn_and_wait` que forzará a que la tarea espere a la finalización de la tarea hija. Para ello, es necesario que el programador controle el número de tareas hijas que tiene la tarea. En cualquier momento una tarea podrá llamar a `wait_for_all` de modo que se bloquee esperando a que todas las tareas hijas hayan finalizado. Una vez finalice una tarea, ésta decrementa el contador de hijos de la tarea padre y es destruida. Por otro lado, existe un mecanismo llamado *Recycling*: si una tarea se marca durante su ejecución utilizando el método `recycle` esta tarea no se destruirá cuando finalice. La tarea es introducida en el *pool* de tareas nuevamente y no se decrementa el contador de la tarea padre. El reciclado es muy útil porque evita la creación y destrucción de nuevas tareas por lo que la sobrecarga introducidas por el mecanismo de gestión de tareas es menor.

Existen tres tipos de tarea:

- Las tareas de tipo *root* no son hijas de ninguna otra. Por tanto, se crean inicialmente y tienen que esperar a que el total de las tareas ejecutadas que dependen de

ellas finalicen para que la ejecución continue. Para poder lanzar una tarea de este tipo hace falta llamar al método `spawn_root_and_wait`.

- Las tareas de tipo *continuation* se ejecutan inmediatamente después de que finalice la tarea actual heredando el mismo padre.
- Las tareas de tipo *child* son hijas de la tarea actual de manera que se encolan en el momento de realizar el *spawn* y se empezarán a ejecutar cuando el thread que ejecuta la tarea actual extraiga una tarea *child* de la cola o bien otro thread la robe por el mecanismo *work-stealing*.

Las listas de tareas TBB permiten agrupar varias tareas para ser lanzadas conjuntamente, ya que de esta manera se consigue mayor eficiencia que mediante un `spawn` individual de cada una de ellas.

La principal característica de TBB es el planificador dinámico que distribuye el paralelismo disponible para mejorar el rendimiento. Este planificador dinámico es en principio transparente al programador aunque tampoco es demasiado complicado modificar su comportamiento reprogramando algunas funciones de la librería TBB. El planificador se inicializa invocando al método `tbb::task_scheduler_init`, el cual crea un conjunto de threads trabajadores. Cuando un thread trabajador es creado, inmediatamente se le asocia una cola de tareas y llama al procedimiento `wait_for_all()`, el cual implementa el planificador de tareas TBB (ver [19]). Este bucle consiste en tres bucles anidados que intentan obtener trabajo a través de tres formas: i) explícitamente por paso de tareas, ii) cogiendo tareas de la cola local y iii) aleatoriamente robando tareas de otras colas (*work-stealing*). Una vez seleccionada la tarea a ejecutar, el bucle más interno del anidamiento es el responsable de ejecutar dicha tarea invocando su método `execute()`. Mientras se está ejecutando la tarea, ésta puede lanzar otras mediante una llamada al método `spawn`, el cual encola la nueva tarea en la cola de tareas del thread/planificador correspondiente.

Como hemos comentado anteriormente TBB cuenta con distintas plantillas de programación paralela para facilitar al usuario la implementación de aplicaciones en sistemas multicore. A continuación vamos a describir algunas de las más importantes y proporcionar un ejemplo de su uso.

#### 1.3.1.1. Parallel_for

Como su nombre indica, esta plantilla implementa un bucle *for* de forma paralela. La plantilla genérica se define mediante la llamada `parallel_for<Range,Body>` donde Range representa el espacio de iteraciones sobre el que se realizará la computación y Body será un objeto de una clase con un método "`operator()`" donde se

especifican las operaciones a realizar para cada iteración. A este objeto se le conoce como "*functor*" usando la terminología de C++. Cuando existen threads disponibles, `parallel_for` ejecutará las iteraciones en paralelo y en un orden no determinista.

Un ejemplo sencillo de utilización de `parallel_for` es el cálculo de la media de N enteros, como el que se muestra en la figura 1.8. Este ejemplo define la función `Parallel_Average` que devuelve en `output[i]` la media de las entradas `input[i-1]`, `input[i]` e `input[i+1]` con $i < n$.

Como vemos, declaramos la estructura `Average` que redefine el `operator()` entre las líneas 7 y 10 de la figura 1.8. Este método `operator()` especifica que operaciones hay que llevar a cabo en el rango de iteraciones que corresponda a cada tarea que concurrentemente colabore en la ejecución del bucle `parallel_for`. En la función `Parallel_Average` creamos el objeto `avg` de la clase `Average` (nuestro *functor*) en la línea 14. A continuación se inicializan los atributos de `avg` para que apunten a los arrays de entrada y de salida. Por último a la plantilla `parallel_for`, en la línea 17, le pasamos un espacio de iteraciones entre 1 y `n` que serán distribuidas siguiendo un esquema `blocked` y el objeto `avg` recién creado.

```
1  #include "tbb/parallel_for.h"
2  #include "tbb/blocked_range.h"
3  using namespace tbb;
4  struct Average {
5    const float* input;
6    float* output;
7    void operator()( const blocked_range<int>& range ) const {
8      for( int i=range.begin(); i!=range.end(); ++i )
9       output[i] = (input[i-1]+input[i]+input[i+1])*(1/3.f);
10   }
11 };
12 // Nota: Leer input[0..n] y escribir  output[1..n-1].
13 void ParallelAverage( float* output, const float* input, size_t n ) {
14   Average avg;
15   avg.input = input;
16   avg.output = output;
17   parallel_for( blocked_range<int>(1, n), avg );
18 }
```

Figura 1.8: Ejemplo de uso de la plantilla `parallel_for` de TBB.

#### 1.3.1.2. Parallel_do

Esta plantilla sirve para expresar trabajo en paralelo cuando no se conoce a priori el número de iteraciones a ejecutar (por ejemplo en un "while"). Es necesario indicar el

Body (como en el `parallel_for`) que realizará el trabajo en paralelo sobre un iterador que se le pasa como parámetro. Además, con el método `feeder.add()` podremos dinámicamente añadir más trabajo al espacio de iteraciones. Para lograr que un problema con esta plantilla escale es necesario que el grano de tarea sea al menos de 100.000 ciclos de reloj. En otro caso, el overhead interno será muy grande en comparación con la carga computacional.

#### 1.3.1.3. Pipeline

Un *pipeline* representa una computación basada en varios filtros en serie que procesan un *stream* de datos. Cada filtro puede operar de manera diferente: en paralelo, en serie, o en desorden. Un pipeline contiene uno o más filtros. Cada filtro es una clase que hereda de la clase `filter`. Es necesario sobrescribir el método `filter::operator()` para especificar la acción que cada filtro realizará sobre cada uno de los elementos del *stream* de datos.

```
1 // Inicializar el planificador de TBB
2  tbb::task_scheduler_init init (NTHREAD);
3
4 // Crear el objeto pipeline
5  tbb::pipeline pipeline;
6
7 // Crear la primera etapa del pipeline
8  MyInputFilter input_filter(input_file);
9  pipeline.add_filter(input_filter);
10
11 // Crear la segunda etapa del pipeline
12  MyTransformFilter transform_filter;
13  pipeline.add_filter(transform_filter);
14 ...
15
16 // Ejecutar el pipeline
17  pipeline.run(n_tokens);
18
19 // Liberar la memoria ocupada por el objeto pipeline
20  pipeline.clear();
```

Figura 1.9: Ejemplo de uso de la plantilla `pipeline` de TBB.

En la figura 1.9 presentamos un ejemplo de uso de la plantilla `pipeline` de TBB. La línea 2 es opcional, pero se puede usar para definir el número de threads lógicos que deseamos que ejecuten las tareas del pipeline. Si esta línea no se especifica se usarán tantos threads como cores estén disponibles en el sistema. En la línea 5 se crea una ins-

tancia de tipo *pipeline* que iremos configurando posteriormente con las distintas etapas. Suponiendo que disponemos de una clase para cada etapa del pipeline (en nuestro ejemplo las clases `MyInputFilter` y `MyTransformFilter`), el siguiente paso, en las líneas 9 y 13, es crear instancias de cada uno de los filtros y añadirlas con el método `pipeline::add` a la instancia de *pipeline*. El *pipeline* estará ya completo y tan solo se debe invocar al método `pipeline.run` para poder ejecutarlo (línea 17). Este método `pipeline.run` contiene un parámetro configurable, `n_tokens`, que indica el número de elementos del *stream* de datos que pueden estar siendo procesados a la vez entre todas las etapas del pipeline. Por ejemplo, si `n_tokens`=5, el pipeline solo admite 5 elementos "en vuelo" durante la ejecución del pipeline y por tanto, si ya hay 5 elementos en ejecución, hasta que no termina uno en la última etapa no puede entrar otro por la primera. Conforme aumentamos el valor de esta variable, la concurrencia irá aumentando, pero a costa de un mayor consumo de memoria ya que el tamaño de las colas entre las etapas puede aumentar.

En la figura 1.10 se muestra el esquema de una ejecución pipeline y en la figura 1.11 se resume el kernel del método `execute` para la plantilla pipeline. Discutimos los detalles de estas figuras a continuación.

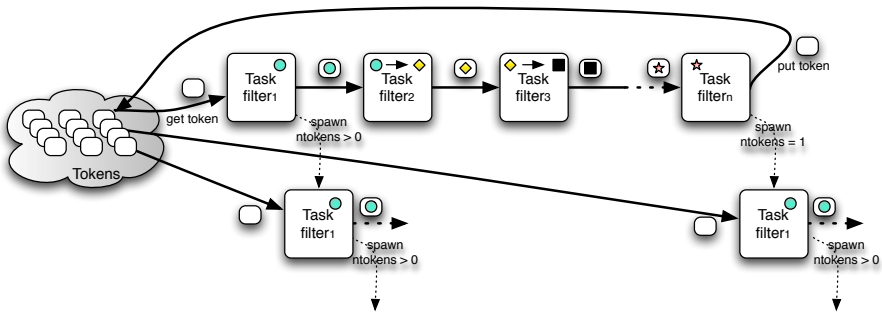



Figura 1.10: Diagrama del funcionamiento del método `execute`() de la plantilla `pipeline`.

En la plantilla `pipeline` de TBB, un token representa un elemento, item o tarea que tiene que atravesar todas las etapas. Por cada elemento de entrada se crea un nuevo token. En una etapa en serie, cada token puede ser procesado en orden o en desorden, mientras que en una etapa paralela pueden ser procesados de modo concurrente. Como se ha mencionado anteriormente, para evitar el consumo no deseable de recursos por la etapa más rápida en el *pipeline*, `n_tokens` especifica el número máximo de tokens que pueden estar en vuelo. Una vez que limitamos este parámetro, la clase `pipeline`

nunca crea un nuevo token en la etapa de entrada hasta que otro token es destruido al final de la etapa del *pipeline*. Como vemos en la figura 1.9, el programador que use esta plantilla de `pipeline` no tiene que controlar la cola de elementos y la inicialización, ni preocuparse por encolar o desencolar elementos.

```
1 task * stage_task::execute(){
2   if (first_entry && --input_tokens>0){
3      spawn_sibling(stage_task); ...}
4
5   item=my_filter.function(item /*anterior*/); //ejecuta la etapa
6   my_filter=my_filter->next_filter_in_pipeline;
7
8   if(my_filter){ //si no es la última etapa
9       ...
10      recycle_as_continuation(); next=this;  //Recicla la tarea
11      return next;}
12  }else {   //último filtro completado
13     input_tokens++;   // despierta un nuevo token/item
14     ...
15     return NULL;
16  }
17 }
```

Figura 1.11: Método `execute()` de la plantilla `pipeline`.

Como se muestra en las figura 1.10 y 1.11, cada tarea empieza su ejecución en la primera etapa del pipeline, en la que, si el número de tokens en vuelo es menor que `n_tokens` (`input_tokens` $>$ 0), se lanza una tarea hermana (ver líneas 2 y 3 de la figura 1.11 o el estado $filter_1$ del esquema de la figura 1.10). En cualquier caso, la tarea ejecuta la función del filtro actual a la que se le pasa un item de la etapa anterior y devuelve un item ya procesado para la etapa siguiente (ver líneas 5 y los objetos que se pasan entre etapas en el esquema: un círculo entre las etapas 1 y 2, un rombo entre 2 y 3, etc.). A continuación se avanza al siguiente filtro en la lista de filtros como se ve en la línea 6 del código. Si no hemos alcanzado el último filtro del pipeline (ver línea 8), en vez de terminar la tarea actual y crear una nueva para el próximo filtro, en la línea 11 de la figura 1.11, la tarea se reciclará como tarea "de continuación" y se devolverá un puntero a ésta al planificador para que sea la misma tarea la que pase a ejecutar la siguiente etapa del pipeline. El mecanismo de reciclado tiene ventajas como explican los autores en [71] y [19] porque tiene varias implicaciones en el rendimiento debido a un uso eficiente de la memoria caché: cuando una tarea ha terminado de procesar un dato, el thread/core donde se ha ejecutado tiene probablemente ese dato en la memoria caché. Si la tarea se recicla para ejecutar la siguiente etapa, la tarea reciclada procesa el dato en el mismo thread/core, por tanto se reduce el número de fallos de caché. Por otro lado, en

el caso de que un token haya alcanzado la última etapa del pipeline (línea 12), se libera y se devuelve `NULL` (línea 16). En ese momento el planificador intentará extraer otra tarea desde la cola local en orden LIFO. Si no tiene éxito, se procederá a robar una tarea a otro thread tal y como hemos estudiado.

### 1.3.2. OpenMP 3.0

OpenMP se ha convertido en una librería estándar para la programación en sistemas de memoria compartida. Dispone de un interfaz muy simple para desarrollar aplicaciones paralelas. Nació en 1990 con la necesidad de estandarizar las diferentes directivas relacionadas con el paralelismo, como podemos ver en la figura 1.12.

OpenMP se estructuró alrededor de los bucles paralelos y esto fomentó su uso para aplicaciones numéricas. OpenMP está preparado para aplicaciones escritas en C/C++ o Fortran en cualquier arquitectura, incluyendo Unix o Windows. Una de las principales características es que OpenMP es escalable y sirve para implementar desde aplicaciones de escritorio hasta aplicaciones para supercomputadores. En OpenMP, un thread maestro se divide en distintos threads esclavos. Los threads se ejecutan concurrentemente en los distintos procesadores ejecutando el trabajo en paralelo. Durante los últimos años, las aplicaciones modernas se han vuelto más complejas cada vez. Esto implica que ya no es suficiente con los tradicionales bucles paralelos para conseguir la máxima eficiencia y aprovechamiento de las nuevas arquitecturas. Por eso a partir de la versión 3.0, OpenMP implementa un modelo basado en tareas.

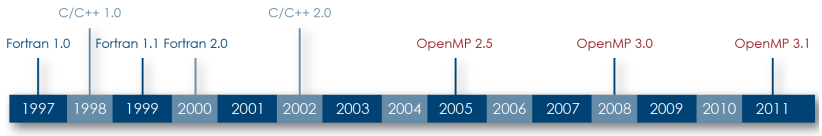



Figura 1.12: Cronograma histórico de la evolución de OpenMP.

OpenMP es una API (Application Program Interface) basada en directivas. El programador añade las directivas al código y estas serán interpretadas por un compilador que soporte OpenMP, de forma que se generará un código que podrá ser ejecutado en paralelo. Unas de las características fundamentales de OpenMP es que se puede compilar un código escrito con directivas OpenMP para obtener tanto la versión secuencial como la paralela, simplemente compilando sin o con el *flag* necesario para ignorar o interpretar las directivas OpenMP, respectivamente. Las directivas de OpenMP son utilizadas para

varios propósitos: crear una región paralela, dividir bloques de trabajo entre los threads, realizar bucles paralelos y sincronizar los distintos threads. Ejemplos de directivas son: `omp_parallel`, `omp_parallel_for`, `omp_task`, etc.

OpenMP incluye una gran cantidad de rutinas que permiten tener un mayor control de la ejecución del código paralelo. Estas rutinas están diseñada con el objetivo de configurar el número de threads que se quiere ejecutar en cada momento, obtener el identificador de cada thread, comprobar si hay paralelismo anidado, calcular el tiempo de ejecución, etc. Para configurar la ejecución de cada programa existen una serie de variables de entorno que se pueden personalizar. La más importante es OMP_NUM_THREADS, que sirve para configurar la cantidad de threads que ejecuta un programa en OpenMP. Otros parámetros configurables son por ejemplo los que permiten especificar como se dividen las iteraciones de un bucle, habilitar y deshabilitar el paralelismo anidado, cuantos niveles de anidamiento se permiten, etc.

#### 1.3.2.1. Tareas en OpenMP

Hasta la versión 2.5 de OpenMP, la API estaba basada en threads. El modelo se basaba en el paradigma de ejecución paralela *fork-join*, donde todos los threads tienen acceso a la memoria compartida. Sin embargo, a partir de la versión 3.0 de OpenMP, la API fue modificada basando su construcción en el modelo de tareas. Las directivas OpenMP distribuyen el trabajo en los threads, pero la diferencia con las versiones anteriores es que a partir de ahora cada thread ejecuta implícitamente tareas. Además, introduce nuevas directivas que permiten al usuario crear sus propias tareas, muy útil para expresar paralelismo no estructurado. Todo lo que esté dentro del ámbito de una directiva `omp_task` se ejecutará como una tarea. El mecanismo de reciclado de tareas no existe implícitamente en OpenMP. Para simularlo, podemos crear un procedimiento que se encargue de realizar el trabajo de una tarea. La directiva `omp_task` encierra la llamada a este procedimiento creando así la tarea. En el momento en el que queremos reciclarnos, realizamos una llamada recursiva a este procedimiento sin utilizar la directiva, evitando así crear una nueva tarea. Para poder sincronizar las tareas se utiliza la directiva `taskwait`, que se encarga de esperar a todas las tareas activas.

### 1.3.3. Intel Cilk Plus

Cilk es una extensión del lenguaje C/C++. Intel proporciona una versión de este lenguaje denominada Intel Cilk Plus. El lenguaje Cilk se desarrolló en 1994 en el MIT y es también un lenguaje basado en el modelo de programación de tareas. En este lenguaje el programador es responsable de implementar el código paralelo, de identificar los ele-

mentos que deben ser accedidos en exclusión mutua, así como los puntos donde se deben sincronizar las distintas tareas. Al ser Intel Cilk Plus una extensión de C y C++, permite de manera rápida y fácil mejorar el rendimiento de programas en sistemas multicores ya que con sólo tres palabras claves se pueden escribir una versión paralela. Eliminar o ignorar estas palabras clave permite a posteriori compilar un ejecutable secuencial. Estas palabras clave soportadas por el lenguaje son las siguientes:

- `Cilk_sync`: Permite sincronizar tareas que hayan sido lanzadas en un punto del código.
- `Cilk_spawn`: Es la encargada de crear y lanzar una nueva tarea.
- `Cilk_for`: Paraleliza un bucle for.

Un ejemplo de cómo utilizar Cilk se puede ver en la figura 1.13 donde se implementa para calcular en paralelo números de Fibonacci. Creamos con `cilk_spawn` las nuevas tareas y sincronizamos con `cilk_sync`.

```
1 cilk int fib (int n) {
2  if (n < 2) return n;
3  else{
4   int x, y;
5   x = cilk_spawn fib(n-1);
6   y = cilk_spawn fib(n-2);
7   cilk_sync;
8   return (x+y);
9  }
10 }
```

Figura 1.13: Ejemplo del cálculo en paralelo de un número de Fibonacci utilizando Cilk.

### 1.3.4. CnC

*Intel Concurrent Collections* para C++ proporciona un mecanismo que facilita la escritura de programas C++ para ejecutar en sistemas multicore. En CnC no hay necesidad de pensar técnicas a bajo nivel como primitivas para utilizar threads o paso de mensajes. Tampoco es necesario pensar a un nivel un poco más alto como son las tareas, pipeline o bucles. CnC mantiene por separado la semántica de una aplicación y su implementación. A diferencia de los anteriores paradigmas, CnC funciona perfectamente en sistemas de memoria compartida y distribuida. Además, los mismos códigos funcionan también tanto en Windows como en Linux y las aplicaciones generadas son relativamente eficientes. La idea básica de CnC es facilitar la programación al usuario. Este no tendrá que

pensar qué debe ir en paralelo, sino que simplemente tendrá que especificar las dependencias semánticas de su algoritmo y las restricciones necesarias para que el resultado sea correcto. El modelo permite al programador especificar a alto nivel computacional los pasos, incluyendo las entradas y salidas pero sin expresar cuándo o dónde deberían ser ejecutadas. Una especificación completa en CnC es un grafo donde los nodos pueden ser de distinto tipo y los arcos representan un productor, un consumidor o relaciones de control. CnC está desarrollado utilizando TBB, por tanto a bajo a nivel es un modelo de programación basado en tareas que utiliza una estrategia de *work-stealing*.

Un ejemplo de cómo utilizar CnC se encuentra en la figura 1.14. En esta librería, el método `put` de las líneas 10 y 14 es equivalente al `spawn` de TBB.

```
1  int Operation::execute(const par & t, simple2D_context & c) const
2  {
3    int i = t.first;
4    int j = t.second;
5
6    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
7
8    if (i < n-1)
9      if (--counter[i+1][j] == 0)
10       c.ElementTag.put(par(i+1, j));
11
12   if (j < n-1)
13     if (--counter[i][j+1] == 0)
14       c.ElementTag.put(par(i, j+1));
15
16   return CnC::CNC_Success;
17 }
```

Figura 1.14: Ejemplo de un problema wavefront utilizando CnC.

# 2 Estudio del modelo basado en tareas para pipeline

Este capítulo se centra en la programación de aplicaciones de tipo pipeline utilizando el paralelismo de tareas. En un *pipeline*, la aplicación se divide en una secuencia de filtros o etapas. Los filtros son regiones de código que pueden exhibir otros tipos de paralelismo (paralelismo de datos o tareas). Hemos elegido dos aplicaciones incluidas en el conjunto de Benchmarks PARSEC, ferret y dedup, como representativas del paradigma de programación paralela del tipo pipeline. Ferret implementa una búsqueda de similitud de imágenes y dedup comprime un *stream* de datos utilizando un método de deduplicación.

Consideramos el paralelismo de pipeline como un patrón emergente de programación y por ello estamos interesados en proveer modelos, herramientas y una guía a los programadores sobre la escalabilidad de las aplicaciones basadas en este patrón. Las aplicaciones paralelizadas utilizando el patrón pipeline son muy sensibles al balanceo de carga. Para una mejor eficiencia, todas las etapas de pipeline deben mantenerse ocupadas durante la ejecución. Esto implica que el programador debe dividir el trabajo en bloques de trabajo bien balanceados a través del pipeline o utilizar un sistema que disponga de administración y reserva dinámica de los recursos (y que no introduzca demasiado overhead). Cuando utilizamos pipeline encontramos otro cuello de botella típico, la entrada y salida de datos que se localiza normalmente al final del pipeline. Como explicaremos a continuación, ferret está limitado por la entrada, mientras que dedup está limitado por la etapa de salida la cual tiene que ser serializada para devolver un único fichero. Optimizar estas etapas requiere modificaciones en los algoritmos, mientras que el problema de balanceo de carga puede ser corregido utilizando métodos automáticos como el *work-stealing* [9, 17].

Han existido diferentes intentos para proporcionar un alto nivel de abstracción para expresar paralelismo de tipo pipeline. Algunos de esos esfuerzos confían en lenguajes especializados de programación: Brook [11] , StreamIt [78] o StreamC/KernelC [20]. Sin embargo, cada vez es más difícil adoptar un nuevo lenguaje. Debido a esto, las librerías de paralelización basadas en tareas como TBB [43] o Microsoft's Task Parallel Library (TPL) en .NET [46] ofrecen alternativas atractivas para utilizar códigos heredados con un pequeño esfuerzo de codificación.

Además del constructor general de tareas, TBB incluye constructores o plantillas de pipeline para expresar este paradigma de manera más sencilla. Nos centraremos en la plantilla de pipeline, ya que la mayoría de las aplicaciones de *stream* pueden ser expresadas con una configuración lineal sin ciclos [74] y este tipo de configuración es la que pueden ser fácilmente expresada con la plantilla de pipeline.

En la primera mitad del capítulo profundizaremos en los siguientes puntos:

- Una caracterización detallada de los dos códigos del conjunto de benchmarks PARSEC que utilizan paralelismo de pipeline. En nuestro estudio, identificamos las principales dificultades que limitan la escalabilidad de los códigos y proponemos algunas soluciones.
- Una comparación del rendimiento de las diferentes implementaciones de los dos benchmarks: las versiones originales en Pthreads y una versión utilizando work-stealing implementada en TBB para cada código.

Y en la segunda mitad realizaremos optimizaciones a la plantilla de TBB para resolver los problemas de implementación que encontramos al programar el código dedup:

- La introducción y descripción de un nuevo tipo de filtro para la plantilla de TBB pipeline. Este nuevo tipo de filtro, llamado *multioutput*, simplifica la codificación no trivial de la estructura pipeline paralelo en la cual algunas de las etapas produce más elementos de salida que elementos entrantes.
- Un análisis comparativo de nuestra implementación basada en el filtro multioutput con otras aproximaciones basadas en los filtros estándar de pipeline de TBB. Discutimos los pros y los contras de cada implementación.
- Una descripción analítica para los modelos de las diferentes implementaciones propuestas. Nuestros modelos están basados en teoría de colas y tienen el principal objetivo de ayudarnos a un mejor entendimiento del uso de los recursos del sistema en cada caso. Validamos el modelo analítico para cada implementación, usando dedup como caso de estudio.

- Una evaluación del rendimiento y discusión de las fuentes de overhead para cada implementación.

## 2.1. Códigos pipeline

Nuestro primer objetivo es estudiar el comportamiento de la implementación original de ferret y dedup en Pthread. Para analizar el rendimiento, hemos compilado y ejecutado los códigos en un multiprocesador de memoria compartida. La máquina elegida es un HP9000 Superdome con 64 dual core Itanium 2 9140 (128 cores total), a 1.6GHz. Cada core tiene la siguiente jerarquía de memoria: 16KB L1I + 16KB L1D, 1MB L2I + 256KB L2D y una caché L3 unificada de 9MB. La memoria principal compartida tiene 380 GBs y la máquina está conectada por 14 buses SCSI I/O a 40TB en un RAID de 5 discos duros. El sistema operativo es Linux SLES 10 SP2 con kernel 2.6.16 y los compiladores utilizados son Intel icc 10.1 y gcc 4.1.2.

### 2.1.1. Búsqueda de similaridad: ferret

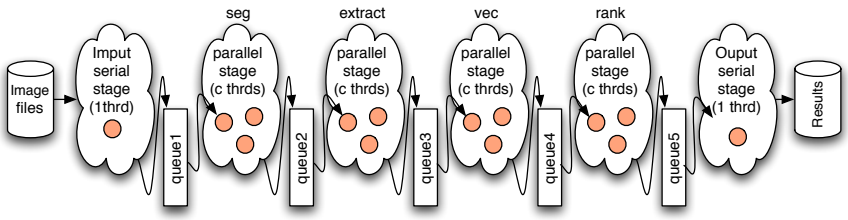



Figura 2.1: Configuración del pipeline en `ferret`. Las etapas de entrada y salida son serie, pero las etapas centrales son paralelas, cada una con $c$ threads (3 en este ejemplo).

El kit de herramientas Ferret es utilizado para realizar una búsqueda de contenido de audios, imágenes, vídeo, formas 3D y datos genómicos. La aplicación `ferret` incluida en PARSEC es una instancia particular del conjunto Ferret configurado sólo para búsqueda de similaridad entre imágenes [6]. Como podemos ver en la figura 2.1, `ferret` implementa un pipeline de 6 etapas. La primera y la última son etapas serie de entrada y salida, donde respectivamente, un thread está encargado de leer un conjunto de imágenes para las cuales `ferret` irá buscando similitudes en una base de datos, y otro thread irá escribiendo en el disco para cada dato de entrada, la lista con los nombres de

los resultados de similitud entre las imágenes. Las cuatro etapas centrales son paralelas: `seg()`, `extract()`, `vec()`, y `rank()`.

Cada una tiene su propio pool de threads locales que puede ser configurado con $c$ threads paralelos (en la figura 2.1 hemos ilustrado tres threads por etapa). Los datos o elementos van desde una etapa a la siguiente y son guardados temporalmente en colas software (desde 1 a 5), todas ellas con capacidad para 20 elementos (tamaño por defecto: $Queue\_size$). De los diferentes conjuntos de entradas disponibles en PARSEC para el benchmark `ferret`, en [57] seleccionamos el más grande (`native`). Para este conjunto de entradas las versión secuencial tarda 760 segundos (utilizando gcc -O3) y 437 segundos (utilizando icc -O3).

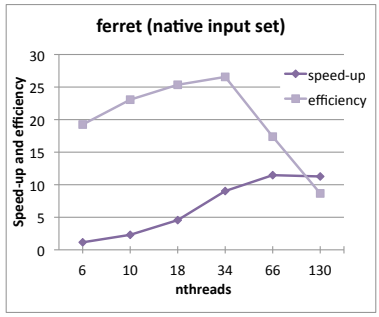



Figura 2.2: Código original de `ferret`: speedup y eficiencia.

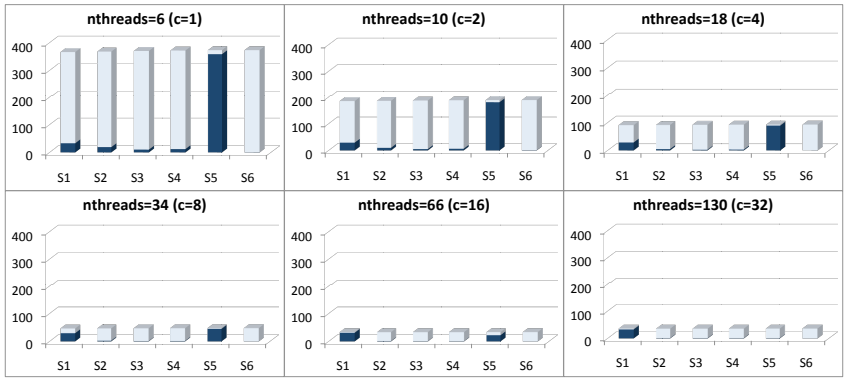



Figura 2.3: Código original de `ferret`: desglose de tiempos (en segundos): para cada etapa, azul oscuro representa el trabajo útil mientras el azul claro representa el tiempo de espera de los threads.

En la figura 2.2 podemos ver el speedup y la eficiencia obtenida cuando cambiamos el parámetro de entrada $c$ en `ferret` entre 1 y 32 (así el número total de threads, $nthreads$, varía entre 6 y 130, siendo calculado como $nthreads = c \times 4 + 2$). En esta figura, el máximo speedup y eficiencia son 12,5 y 27 %, respectivamente. Para explicar los motivos de estos pobres resultados de eficiencia y escalabilidad nos referimos a la figura 2.3. En esta figura se ilustra el tiempo utilizado por cada etapa (numeradas del 1 a 6) para llevar acabo su trabajo (en azul oscuro) y el tiempo de espera de cada etapa para leer/escribir en las colas del pipeline (en azul claro).

Es evidente que la quinta etapa (`rank`()) es la más lenta con diferencia cuando $c = 1$. Las etapas de 1 a 4 no puede proceder porque las colas 1 a 4 están llenas. `rank`() es demasiado lenta cogiendo elementos de `queue4` y este efecto se propaga hacia atrás. Al aumentar el número de threads, el tiempo de trabajo en paralelo se reduce, pero con $c = 16$ ($nthreads = 66$), la etapa de entrada en serie comienza a ser un cuello de botella. En el estudio [57] se comprueba que el tiempo de ejecución está limitado por la etapa más lenta del pipeline.

### 2.1.2. Compresión de imágenes: dedup

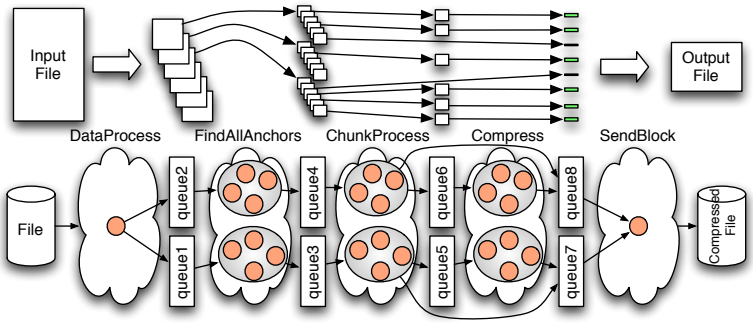



Figura 2.4: Configuración del pipeline en `dedup` con $c = 8$ threads por etapa paralela del pipeline.

El kernel `dedup` logra una ratio alta de compresión de un *stream* de datos combinando una compresión global y una local. Para llevar acabo la compresión, también llamada *de-duplication*, el código implementa cinco etapas de pipeline. Como en `ferret`, la primera, `DataProcess`() y la última etapa, `SendBlock`(), son secuenciales. Las etapas centrales se ejecutan en paralelo: `FindAllAnchors`(), `ChunkProcess`() y

`Compress`(). Básicamente, `DataProcess`() genera unidades de trabajo independientes (chunks) las cuales son encoladas secuencialmente. `FindAllAnchors`() (la etapa siguiente), separa cada dato de grano grueso en distintos chunk de bloques de grano fino de 512 bytes de media. `ChunkProcess`() únicamente identifica cada uno de esos bloques por computación de su SHA1 checksum y entonces en una tabla hash global si el bloque no fue registrado, pasa a la etapa `Compress`(). En otro caso, el bloque es clasificado como duplicado (ya comprimido) y directamente enviado a la etapa de salida `SendBlock`(). Este bypass de la etapa `Compress`() ha sido ilustrado en la figura 2.4.

Hasta aquí, las dos principales diferencias en la configuración del pipeline `dedup`, comparado con `ferret` son que i) hay una etapa que genera más elementos de salida que elementos de entrada (`FindAllAnchors`()); y ii) algunos elementos pueden omitir una etapa del pipeline (`Compress`()). Además, en la figura 2.4 podemos notar una diferencia adicional: para evitar la contención en el acceso a colas, éstas están compartidas por un máximo de 4 threads, así que hay más colas entre las etapas.

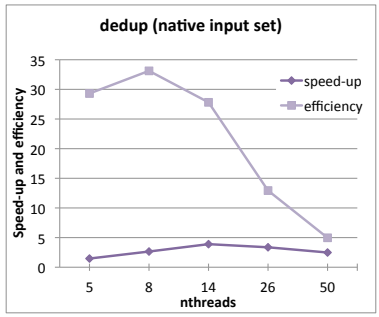



Figura 2.5: Código original de `dedup`: speedup y eficiencia.

La entrada más grande disponible en PARSEC para el benchmark `dedup`, `native`, consiste en un archivo ISO de 672 MB. Elegimos la opción de no precargar el fichero de entrada, para permitir *streaming* real de los datos de entrada en el pipeline. Para este caso, la versión secuencial tarda 91,8 segundos (gcc -O3) y 89,37 segundos (icc -O3).

En la figura 2.5 podemos ver el speedup y la eficiencia obtenida cuando cambiamos el número de threads por etapa, c, entre 1 y 32 (ahora el número total de threads, $nthreads$, esta entre 5 y 98, siendo $nthreads = c \times 3 + 2$). En este código el tamaño por defecto, $Queue\_size$, es un millón , así que las colas nunca se llenan. De cualquier modo, y como en `ferret`, el speedup y la eficiencia son muy bajas debido al alto desbalanceo de carga entre las etapas. Aquí, la etapa más lenta es `Compress` y la salida es un cuello de botella a partir de $c = 5$ (17 threads) como observamos en la figura 2.6.

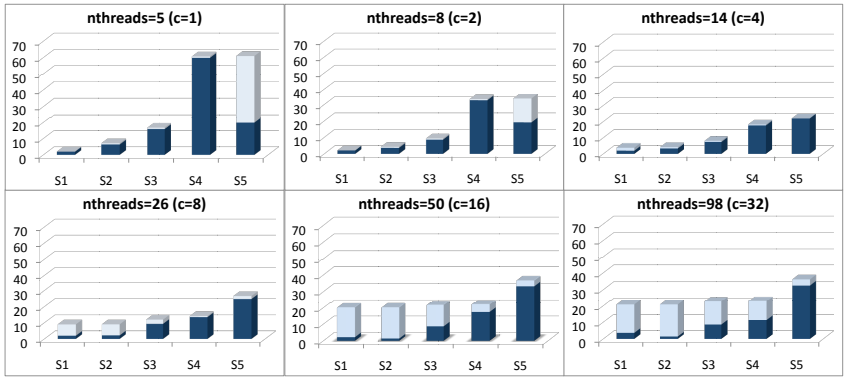



Figura 2.6: Código original de `dedup`: desglose de tiempos (en segundos): para cada etapa, azul oscuro representa el trabajo útil mientras que el azul claro representa el tiempo de espera.

### 2.1.3. Algunas soluciones

Para abordar estos problemas (el rendimiento y el desbalanceo de carga) en [57] se propusieron dos alternativas i) colapsar las etapas en paralelo en un sola etapa, ii) utilizar un planificador dinámico de trabajo compartiendo la carga a través de las diferentes etapas.

La primera solución, colapsar etapas, aunque puede funcionar para `ferret` o `dedup`, podría no ser aplicable al caso general. Si hay una etapa secuencial entre dos etapas en paralelo, entonces al colapsar las etapas en serie con alguna de las paralelas supondría la serialización de la nueva etapa colapsada.

Hemos estudiado dos caminos para implementar el planificador dinámico de trabajo en un pipeline paralelo. La primera opción supone *oversubscription* de threads por etapas. *Oversubscription* como vimos en el capítulo inicial significa que hay más threads lógicos que thread físicos disponibles (cores). En este caso, el sistema operativo puede mantener los cores físicos trabajando cuando un thread lógico esté bloqueado.

Esta técnica puede ser considerada como de planificación semi-dinámica. Los threads preparados están lanzados dinámicamente por el sistema operativo pero el número de threads por etapas pipeline está fijado estáticamente en tiempo de compilación. Sin embargo, recurrir a *oversubcription* puede provocar la aparición de tres fuentes de overhead: *cambio de contexto* (se interrumpe la ejecución de un thread en un core por la ejecución de otro thread), *cache cooling* (al cambiar el contexto, el nuevo thread que se ejecuta en el core puede reemplazar los datos de la memoria cache) y *lock preemption*

(un thread puede marcar el ritmo de ejecución, por ejemplo, éste coge un lock pero no lo libera. Se produce un cambio de contexto y el nuevo thread intenta coger el lock pero se bloquea. Hasta que el thread anterior no vuelva a ejecutarse para liberar el lock se producirá un bloqueo.) [71]. Podemos evitar *oversubscription* mediante la segunda opción que obtiene balanceo dinámico confiando en el paradigma work-stealing [8]. Esto se puede lograr teniendo una gran cantidad de tareas o unidades de trabajo para cada thread o core. Si un thread o core ha completado todas sus tareas, puede intentar robar alguna tarea de un core o thread vecino. Precisamente, esta idea forma parte de la librería de paralelización TBB [71] y debido a su gran potencial le dedicamos más atención en el capítulo para estudiar la solución a estos problemas.

## 2.2. TBB pipeline template

Una de las principales metas declaradas por TBB es reducir significántemente el problema del desbalanceo de carga a través de la técnica *work-stealing* [19]. El entorno de programación de TBB anima a los programadores a expresar la concurrencia en términos de tareas en vez de threads.

Portar la configuración del pipeline `ferret` a TBB no fue demasiado problemático. Sin embargo, `dedup`, fue más laborioso ya que impedía una traducción directa a TBB. Como dijimos en la introducción a `dedup`, una característica especial de este código es que el número de elementos que alcanza la etapa `FindAllAnchors`() es diferente al número de elementos que abandonan esa etapa. Esta es una cuestión problemática para TBB, porque una importante restricción en esta librería es que el número de tokens que abandonan una etapa siempre debe ser igual al número de tokens que llegó a dicha etapa. Debido a esta restricción, en la implementación de TBB no pudimos incorporar la etapa de salida en serie del filtro TBB. En vez de eso, creamos una cola individual y un thread dedicado, utilizando Pthreads, para llevar a cabo la escritura en el fichero de salida. Otra consecuencia de esa misma restricción es que esa etapa `FindAllAnchors`() y las sucesivas etapas (`ChunkProcess`() y `Compress`()) no pueden ser asignadas a diferentes filtros del pipeline en TBB. Así que las colapsamos en un sólo filtro paralelo de pipeline TBB. La etapa de entrada `DataProcess`() ha sido definida como un filtro en serie TBB. Resumiendo, sólo ha sido posible implementar una versión híbrida TBB con una filtro TBB en serie, seguido por un filtro paralelo TBB, seguido por una etapa Pthreads serie (`SendBlock`()).

Nos gustaría mencionar que el trabajo [57] explora también otras posibilidades para implementar estos códigos eficientemente en una arquitectura multicore. Desde un nivel de grano grueso de los bucles externos, ambos códigos son altamente paralelizables, intentándose también una implementación completa de esos bucles. Sin embargo, la na-

turaleza del streaming de las aplicación dificultó nuestro diseño. Por ejemplo, portamos esos códigos a OpenMP utilizando la directiva `task` [59], evaluamos la portabilidad de esos códigos a MapReduce de memoria compartida, implementación basada en la reciente librería Phoenix [67]. Sin embargo, en todas las aproximaciones nos encontramos con distintas dificultades como se explica en [57] por lo que descartamos una paralelización completa (DOALL).

En [57] realizamos una modelo analítico para predecir los tiempos de ejecución con distinto número de threads para los casos de ferret y dedup en sus distintas implementaciones. La meta final de nuestros modelos es entender la interacción de recursos, así como el rendimiento conseguido con el work-stealing. Como explicamos en [57], los modelos pueden también servir como una herramienta para configurar el sistema y asignar el tamaño de las colas, el número óptimo de etapas, el número óptimo de tokens en el sistema y establecer el número de threads paralelos en el que puede escalar antes de llegar al cuello de botella. En dicho trabajo realizamos una comparación entre el modelo analítico y las evaluaciones experimentales que expondremos a continuación, viendo como claramente el modelo analítico parece predecir los tiempos reales de ejecución.

## 2.3. Evaluación

En esta sección discutimos en más detalle el rendimiento medido de las diferentes implementaciones que hemos desarrollado para `ferret` y `dedup`. Primero evaluamos la solución de las etapas colapsadas y comparamos las implementaciones de Pthreads y TBB. También calculamos el overhead inducido por TBB. Y, finalmente, discutimos algunas mejoras para aliviar el cuello de botella de entrada/salida. En esos experimentos utilizamos la misma metodología y plataforma que hemos descrito en el capítulo.

### 2.3.1. Colapsando etapas paralelas

En las figuras 2.7 y 2.8 mostramos una comparación del tiempo descompuesto para `ferret` en la versión original de PARSEC (6 etapas) y nuestra versión colapsada (3 etapas), en Pthreads. Podemos ver en la coordenda “x” el número de etapa en el pipeline y en la coordenada “y” el tiempo útil, en gris claro y el tiempo de espera en gris oscuro. Cada barra representa el tiempo para $n$ threads paralelos, con $n = 4 : 32$. Con el mismo número de threads, las versión de 3 etapas mejora la de 6 etapas. Cuando colapsamos las etapas 2, 3, 4 y 5 de la versión de 6 etapas en la etapa 2’ de la versión de 3 etapas, gran parte del tiempo ocioso se traduce en tiempo útil, prácticamente dividiendo por 4 el tiempo de ejecución. `dedup` tiene un comportamiento similar (figuras 2.9 y 2.10).

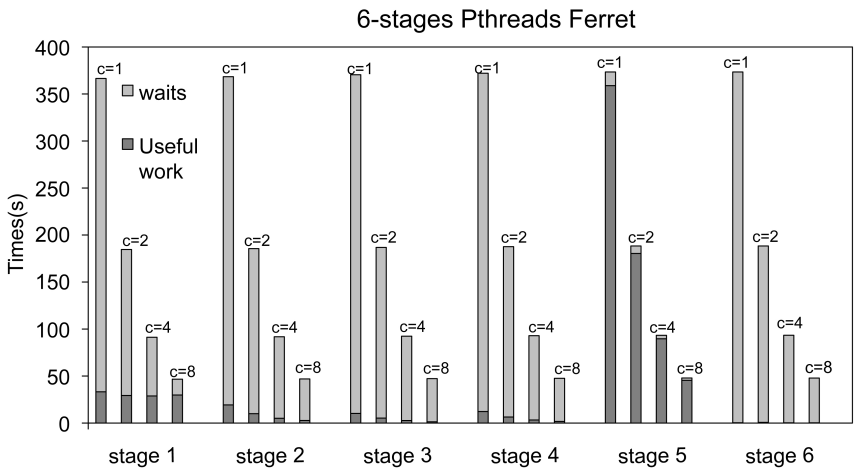



Figura 2.7: Desglose de tiempos (segundos) para `ferret`: versión 6-etapas con Pthreads.

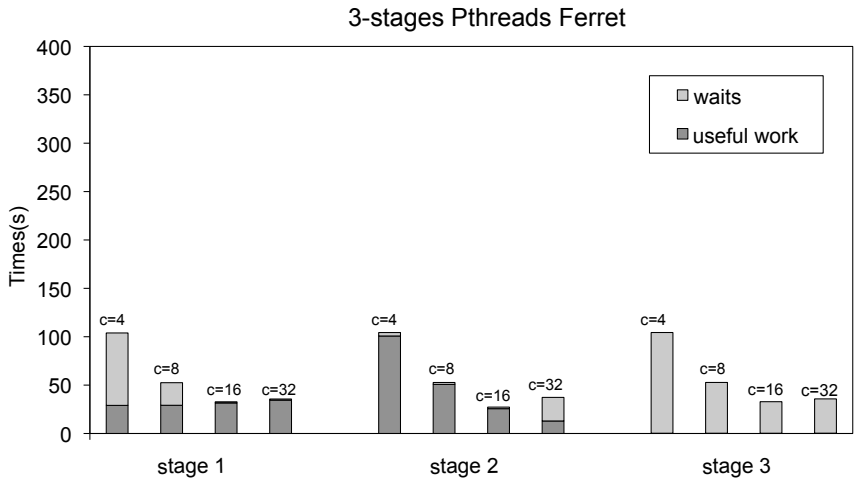



Figura 2.8: Desglose de tiempos (segundos) para `ferret`: versión 3-etapas con Pthreads.

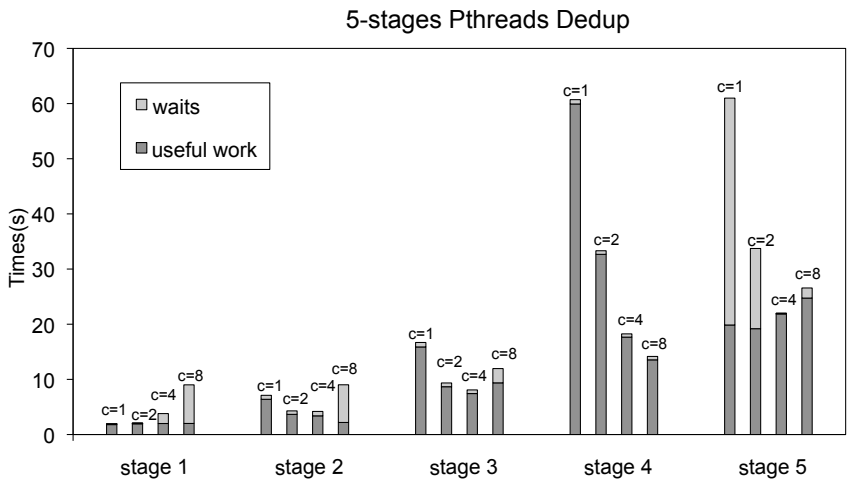



Figura 2.9: Desglose de tiempos (segundos) para `dedup`: versión 6-etapas con Pthreads.

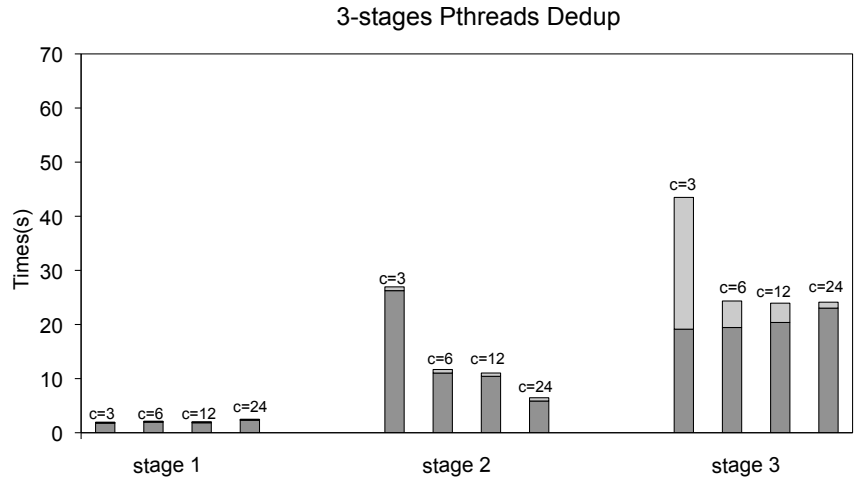



Figura 2.10: Desglose de tiempos (segundos) para `dedup`: versión 3-etapas con Pthreads.

### 2.3.2. Planificador estático vs. work-stealing

Como hemos visto anteriormente, la utilización del hardware en la versión del planificador estático es muy pobre debido al número de threads ociosos. Como mencionamos en la sección 2.1.3, una posible solución para este problema es tener oversubscription de threads; es decir, generar un código multithread donde el número de threads lógicos vivos es mayor que el número de cores. Recordemos que ésta es la opción recomendada en [6]. En este caso, el planificador del Sistema Operativo es el responsable de asignar los threads a los cores para permitir al sistema estar ocupado con trabajo en cualquier etapa en caso de necesidad. Además, estamos interesados en el estudio de la escalabilidad del modelo pipeline así que tenemos que medir el tiempo de ejecución para diferente número de cores. Una opción para conseguir oversubscription con un número parametrizado de cores es utilizar el comando `taskset` de UNIX que confina la ejecución de un proceso (y todos sus threads) en un número dado de cores. De este modo, la comparación entre el tiempo de ejecución en las implementaciones Pthreads y TBB es justa: en Pthreads el sistema operativo controla los cores ocupados asignándole threads a ellos, y en TBB hay solo un thread por core, pero el planificador de TBB asigna tareas a threads ociosos.

| | 6-etapas Pthreads no confinado | | | |
|---|---|---|---|---|
| # threads | # cores | Tiempo (seg.) | Speedup | Eficiencia |
| 6 | 6 | 370,68 | 1,18 | 19,65 % |
| 10 | 10 | 186,69 | 2,34 | 23,41 % |
| 18 | 18 | 94,87 | 4,61 | 25,59 % |
| 34 | 34 | 47,95 | 9,11 | 26,80 % |
| 66 | 66 | 38,41 | 11,38 | 17,24 % |
| | 6-etapas Pthreads confinados | | | |
| # threads | # cores | Tiempo (seg.) | Speedup | Eficiencia |
| 6 | 1 | 438,45 | 1,00 | 99,67 % |
| 10 | 2 | 220,87 | 1,98 | 98,93 % |
| 18 | 4 | 111,88 | 3,91 | 97,65 % |
| 34 | 8 | 69,21 | 6,31 | 78,93 % |
| 66 | 16 | 48,86 | 8,94 | 55,90 % |

Tabla 2.1: Comparación con los threads No confinados vs. Confinados para las 6 etapas de Pthreads del código `ferret`.

En la tabla 2.1 podemos ver el tiempo de ejecución, el speedup (en relación con el tiempo secuencial) y la eficiencia para threads no confinados a cores (no *oversubscription*) y confinados (con *oversubscription*) en la versión 6 etapas Pthreads. En la parte no confinada de la tabla, la primera columna indica el número de threads lógicos (el cual es igual al número de cores en una máquina vacía de 128 cores). Aquí, la eficiencia es relativa a este número de threads lógicos. En la otra mitad de la tabla, tenemos los threads

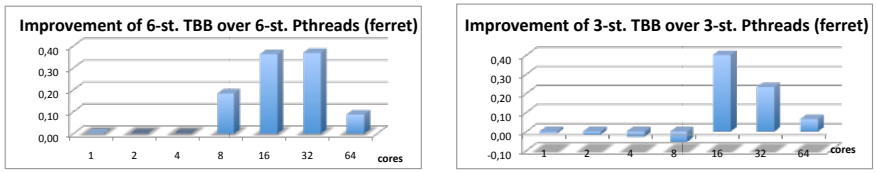
| | 6-etapas TBB | | | 3-etapas TBB | | |
|---|---|---|---|---|---|---|
| # cores | Tiempo (s.) | Speedup | Eficiencia | Tiempo (s.) | Speedup | Eficiencia |
| 1 | 437,65 | 0,77 | 77,00 % | 442,88 | 0,76 | 76,09 % |
| 2 | 221,13 | 1,52 | 76,20 % | 223,5 | 1,51 | 75,39 % |
| 4 | 112,59 | 2,99 | 74,83 % | 113,58 | 2,97 | 74,18 % |
| 8 | 58,5 | 5,76 | 72,01 % | 59,56 | 5,66 | 70,73 % |
| 16 | 35,87 | 9,40 | 58,72 % | 36,98 | 9,11 | 56,96 % |
| 32 | 33,14 | 10,17 | 31,78 % | 33,42 | 10,08 | 31,51 % |
| 64 | 35,66 | 9,45 | 14,77 % | 36,44 | 9,25 | 14,45 % |

Tabla 2.2: Comparación de la implementación TBB de `ferret`: 6-etapas vs. 3-etapas.

confinados. Es claro que usando un número pequeño de cores incrementamos el tiempo de ejecución en la versión confinada, la utilización del hardware ha sido mejorada como podemos ver en la columna de eficiencia.

También hemos discutido que puede ser abordada una solución diferente utilizando un planficador dinámico basado en work-stealing. Hemos implementado `ferret` y `dedup` utilizando la plantilla de pipeline de las librerías de TBB. Empezando con `ferret`, hemos implementado dos versiones de TBB: 6 etapas y 3 etapas. La tabla 2.2 muestra el tiempo de ejecución, el speedup y la eficiencia para esas dos versiones. Como el modelo analítico predice en [57], el número de etapas en la implementación pipeline de TBB prácticamente no afecta al tiempo de ejecución. Gracias a esta planificación dinámica basada en work-stealing, la presencia de etapas desbalanceadas en el pipeline (versión de 6 etapas) no afecta al throughput.

La figura 2.11 muestra la relación entre el tiempo de ejecución en TBB y en Pthreads confinados para `ferret` con 6 y 3 etapas. Se puede apreciar que TBB es más rápido para un número de cores superior a 8 y que para 16 y 32 la mejora es superior al 30 %. Esto es así, porque hay más oportunidades para el work-stealing y el balanceo dinámico de la carga cuando el número de threads es mayor. Las mejoras se degradan cuando llegamos al cuello de botella de entrada, lo cual ocurre antes en la versión de 3 etapas.




(a) (b)

Figura 2.11: Pthreads confinados vs. implementación TBB de `ferret`: (a) 6-etapas y (b) 3-etapas.

Hemos procedido de manera similar para el kernel `dedup`. La figura 2.12 muestra la relación entre la versión de 3 etapas confinada de Pthreads respecto a la versión de etapas de TBB obteniendo una mejora hasta del 63 % para 8 cores en la versión de TBB. Cuando nos aproximamos al cuello de botella, en la etapa de salida, las diferencias se reducen.

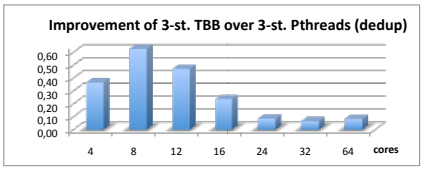



Figura 2.12: Pthreads confinados vs. implementación TBB de `dedup` en 3-etapas.

Las razones por las que TBB es mejor que Pthreads son un mejor aprovechamiento de la localidad y un menor overhead respecto a Pthreads [57].

## 2.4. Mejoras en la plantilla pipeline de TBB

Hasta ahora, en este capítulo hemos estudiado y evaluado el rendimiento de diferentes implementaciones de dos aplicaciones de pipeline [57] pertenecientes al conjunto de benchmarks PARSEC [6]: `ferret` y `dedup`. La implementación original de esos códigos esta basada en implementaciones Pthreads, 6 etapas y 5 etapas respectivamente, con un pool de threads asignados de manera estática a cada etapa. Encontramos que el rendimiento de esos código era muy pobre, principalmente por el desbalanceo de carga. Analizamos en [57] dos soluciones para resolver este problema: i) colapsar etapas y ii) utilizar planificación dinámica basada en work-stealing gracias a utilizar la plantilla de pipeline para TBB. Mientras la primera solución no ofrece una solución general debido a que no se puede aplicar si existe una etapa intermedia en serie, el work-stealing mejora el rendimiento de todas las demás implementaciones en `ferret` y `dedup`. La razón es que la librería de TBB permite al sistema tener un comportamiento que idealmente sería como un pool global de threads trabajadores. También hemos probado que el overhead que introduce esta plantilla es despreciable y el mecanismo de reciclado de tareas que ofrece TBB para pasar elementos a través de los filtros dentro del mismo thread mejora la ratio de aciertos en la cache. Sin embargo, los filtros estándares de TBB tiene una restricción: un filtro solo puede ofrecer un elemento de salida por cada elemento de entrada. Esto limita la implementación del código `dedup` totalmente en TBB, obligando a realizar una implementación híbrida como la descrita en la sección anterior.

A continuación abordaremos este problema y trataremos de mejorar la plantilla de TBB para que una etapa de pipeline pueda enviar más elementos de los que reciba. A este tipo de etapas las llamaremos etapas *multioutput*. Implementaremos una solución en la cual cada vez que un elemento alcanza una etapa multioutput, los nuevos elementos deben ser creados, procesados en paralelo a través de etapas intermedias y eventualmente reordenados debidamente cuando alcancen la etapa en serie. La solución debe garantizar lo siguiente:

- Una interfaz simple y fácil para que el usuario mejore su productividad.
- Las unidades de procesamiento deberán estar bien aprovechadas balanceando la carga si fuera necesario.
- El overhead debe ser pequeño.
- Los recursos consumidos deberán estar controlados. Delimitar la memoria es especialmente importante, ya que esto reduce la probabilidad de thrashing, de condiciones de *out-of-memory* y reduce los fallos de cache.

Hemos desarrollado un nuevo tipo de filtro, llamado *multioutput*, que puede fácilmente integrarse en la plantilla de TBB y que se añade a los filtros serie y paralelo. Describimos los detalles de implementación de este tipo de filtro en la sección 2.5, donde también discutimos y comparamos con otras posibles implementaciones del pipeline paralelo con una salida multiouput utilizando estrategias basadas en tareas. En la sección 2.6 proponemos un modelo analítico para proporcionar algunas señales sobre como optimizar la implementación propuesta. En la sección 2.7 evaluamos en más detalles las distintas implementaciones, a continuación discutimos los trabajos relacionados y finalmente ofrecemos las conclusiones de este capítulo.

## 2.5. Nuevas implementaciones de dedup en TBB

### 2.5.1. Implementación basada en filtros estándar

Como mencionamos anteriormente, el número de elementos que puede alcanzar la etapa `FindAllAnchors`() es distinto al número de elementos que deja esa etapa, lo cual es un problema para una implementación directa de pipeline basada en los filtros estándar de TBB. Un camino para sobreponerse a este problema es colapsar todas las etapas paralelas en la original de pipeline (las etapas `FindAllAnchors`(), `ChunkProcess`() y `Compress`() ) en solo una etapa paralela: `ProcessChunk`().

La etapa de entrada `DataProcess`() es definida con un filtro serie TBB. Sin embargo, como el número de elementos que alcanzan la salida del filtro `ProcessChunk` paralelo es mayor que el número de elementos que entran, entonces la etapa en serie de TBB no puede conectar con la salida del pipeline. Además, la etapa de salida `SendBlock`() tiene que manejar la salida del pipeline. Una posibilidad es crear una cola individual y un thread dedicado utilizando Pthreads para llevar a cabo esa etapa de salida como hemos visto. Con la implementación actual de pipeline de TBB hemos codificado una versión híbrida TBB+Pthreads con un filtro TBB serie de entrada, seguido por un filtro paralelo TBB (el cual comprime las tres etapas paralelas originales), seguido por una etapa serie Pthread. La figura 2.13 muestra el código de la etapa de salida, donde las líneas 2-3 definen la los filtros de TBB, mientras que la línea 5 se refiere a la definición de la etapa de salida `SendBlock`(). Las líneas 15-16 construyen el Pipeline TBB y entonces la línea 20 ejecuta el programa. La línea 18 lanza la etapa de salida creando un thread que ejecuta la función `SendBlock`() definida en la línea 5. A partir de ahora llamaremos a esta implementación Híbrida.

```
1 // Pipeline TBB classes
2 class  DataProcess:tbb::filter {...};
3 class  ProcessChunk:tbb::filter {...};
4 // Pthread  stage
5 void SendBlock(outputfile) {...};
6  ...
7 main( ) {
8 // Start task scheduler
9  tbb::task_scheduler_init init (nthreads);
10 tbb::pipeline pipeline;
11
12 Dataprocess dataprocess (inputfile);
13 ProcessChunk processchunk;
14
15 pipeline.add_filter(dataprocess);
16 pipeline.add_filter(processchunk);
17 // Run the Pthread stage
18 pthread_create(, SendBlock,);
19 // Run the TBB pipeline
20 pipeline.run(ntokens);
21 // Clear the pipeline
22 pipeline.clear ();
23 pthread_join (...);
24   ...
25 };
```

Figura 2.13: Uso de la plantilla pipeline con el filtro estándar de TBB en la implementación Híbrida + Pthreads de `dedup`.

```
// Outer pipeline classes
class  DataProcess:tbb::filter {...};
class ProcessChunk:tbb::filter {
...
protected:
// Nested pipeline classes
class  FindAllAnchors:tbb::filter {...};

class  ChunkProcess:tbb::filter {...};
class  Compress:tbb::filter {...};
class ReassembleBlocks:tbb::filter
     {...};

public:
 void* operator ()( void *token ) {
 tbb::pipeline pipeline;
// Splits chunk into blocks and
     processes
 FindAllAnchors findallanchors(token);
 ChunkProcess chunkprocess;
 Compress compress;
 ReassembleBlocks reassembleblocks();

 pipeline.add_filter(findallanchors);
 pipeline.add_filter(chunkprocess);
 pipeline.add_filter(compress);
 pipeline.add_filter(reassembleblocks);

 // Run the nested pipeline
 pipeline.run(ntokens_in);
 pipeline.clear ();
     ...
  };
};

class SendBlock:tbb::filter {...};
...
```

```
main( ) {
// Start task scheduler
 tbb::task_scheduler_init
      init (nthreads);
 tbb::pipeline p;

 DataProcess dprocess(input);
 ProcessChunk pchunk;
 SendBlock sblock(output);

 p.add_filter(dprocess);
 p.add_filter(pchunk);
 p.add_filter(sblock);

// Run the outer pipeline
 p.run(ntokens);
 p.clear();
}
```

Figura 2.14: Uso de la plantilla pipeline con los filtros estándar de TBB en la implementación Anidada de `dedup`.

Otra solución para implementar el problema `dedup` con filtros estándar de TBB fue propuesta en [70]. Los autores tuvieron que recurrir a pipelines anidados como muestra la figura 2.14, donde mostramos un fragmento de código de su solución denominada *Anidada*. El pipeline interno se encarga de trabajar con los subitems mientras que el exterior lo hace con los chunks (elementos). Los autores necesitan añadir una nueva etapa serie, `ReassembleBlocks`() después de `Compress`() y antes `SendBlock`(),

para rearmar los bloques en chunks antes de pasar los tokens al pipeline externo. Por tanto, ambos pipeline operan en diferente granuralidad de datos (chunks y bloques). En las líneas 13-31 mostramos la construcción y ejecución del pipeline interno anidado.

### 2.5.2. Implementación basada en el filtro multioutput

Otro camino para atajar el problema envuelve un nuevo tipo de filtro para la plantilla TBB de pipeline, el cual representa una de las contribuciones de este capítulo. Este nuevo filtro llamado *multioutput*, se suma a los ya existentes *serial* y *parallel* que son los filtros estándar.

```
1 class StdFilter: public tbb::filter {
2 public:
3  StdFilter();
4  void* operator()(void* item);
5 };
6
7 StdFilter::StdFilter() :
8  tbb::filter(/*serial=*/false)
9 { // Do the initialization }
10
11 void* StdFilter::operator()(void* item)
12 {
13  SomeType *result;
14 // Do some work on item
15 // and return the result
16  return (void *) result;
17 }
```

Figura 2.15: Ejemplo de filtro estándar de TBB.

En la figura 2.15 podemos ver la declaración de un filtro regular de TBB. Por ejemplo en las líneas 1-5 declaramos un filtro llamado `StdFilter`. En el ejemplo, la figura muestra el constructor de la clase y método `operator()`. Este filtro hereda de la clase `tbb::filter` (línea 1), para las cuales el método `operator()` es sobrescrito. Cuando implementamos el constructor de clase, lo invocamos con el parámetro `false`, lo que significa que no es un filtro serie (línea 8). El método `operator()` recibe un objeto de entrada a través de un puntero void (línea 11) y a través de un puntero void devolvemos un objeto con el resultado del trabajo desarrollado por el filtro (línea 16). Obviamente, al ser un filtro TBB se permite un único elemento de salida por cada elemento de entrada.

La figura 2.16 muestra la declaración del filtro multioutput. Como se muestra, declaramos el filtro `MulFilter` (línea 1), y podemos ver que también hereda de la

clase `tbb::filter` como filtro estándar. Sin embargo, cuando implementamos el constructor de la clase para `MulFilter`, especificamos un parámetro adicional, multioutput, el cual es puesto a verdadero (línea 9). Este segundo parámetro especifica que este nuevo filtro es un filtro multioutput. La diferencia más importante del filtro multioutput se encuentra en la definición del método `operator()`: un elemento puede ahora llegar como entrada (línea 12), pero un número desconocido de subitems podrían se devuelto como salida. Para controlar esta situación hemos creado la clase `tbb::object_buffer`. Esta nueva clase está diseñada para mantener los subitems que podrían generarse a la salida de este tipo de filtros. Por esta razón, esta clase tiene dos métodos públicos: el constructor (línea 15) y el método que introduce los subitems (línea 20). El método `operator()` tiene que devolver un puntero a un objeto del tipo `tbb::object_buffer`, donde se guardan los nuevos subitems generados por cada elemento procesado.

```
1 class MulFilter: public tbb::filter {
2 public:
3  MulFilter();
4  void* operator()(void* item);
5 };
6
7 MulFilter::MulFilter() :
8  tbb::filter(/*serial=*/false,
9              /*multioutput=*/true)
10 { // Do the initialization }
11
12 void* MulFilter::operator()(void* item)
13 {
14  tbb::object_buffer *result_buffer;
15  result_buf = new tbb::object_buffer;
16 // Do some work on item and insert
17 // the resulting items in result_buffer
18  while (...) {
19   ...
20   result_buf->put((void *) new_item);
21  }
22 // return all the generated items
23  return (void *) result_buf;
24 }
```

Figura 2.16: Ejemplo del nuevo filtro multioutput.

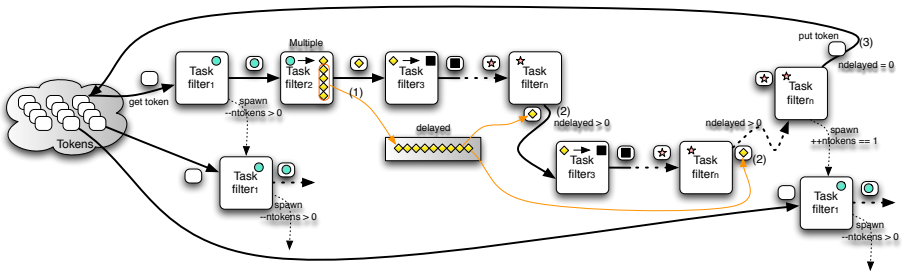
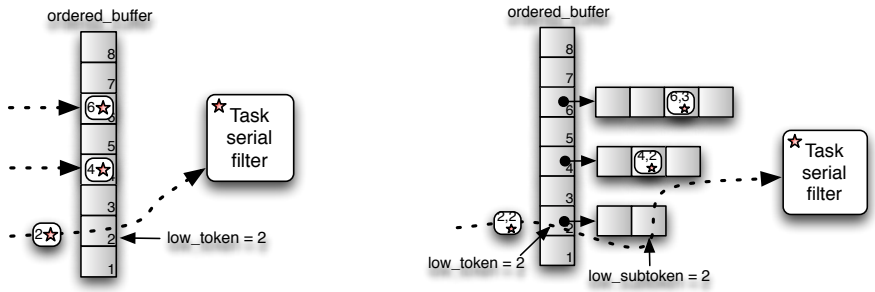



Figura 2.17: Filtro multioutput: esquema de funcionamiento del método `execute()`.

Hemos añadido soporte para el filtro multioutput a través del método `execute()` de la clase tarea del pipeline. En la figura 2.17 podemos ver como trabaja este método. Por ejemplo, asumimos que $\texttt{filter}_2$ es un filtro multioutput. Recordemos que en cualquier pipeline `ntokens` determinan el número de tareas en vuelo que pueden ejecutarse en cualquier momento y que sólo una tarea con un token pueden progresar a través del pipeline. Cuando una tarea ejecuta $\texttt{filter}_2$ es porque tiene en propiedad un token. Por tanto, cuando la tarea termina el trabajo en $\texttt{filter}_2$, genera subitems que son guardados en `tbb::object_buffer`. Obviamente, sólo uno de esos subitems puede progresar al siguiente filtro porque la tarea tiene un solo token. El resto de subitems son encolados y esperan en un buffer llamado `delayed`, donde `ndelayed` representa el número de subitems esperando (ver `(1)` en Fig. 2.17). Asumiendo que una instancia del filtro multioutput genera en media `nsubitems` subitems, y dado que hay `ntokens` en el sistema, entonces encontramos que el tamaño del buffer `delayed` es `ndelayed=ntokens*nsubitems`. En cualquier caso, cuando una tarea alcanza el último filtro, en vez de devolver el token (como hace la implementación original de pipeline), ahora comprueba si hay subitems esperando que todavía necesitan ser procesados (`ndelayed` >0). En este caso, dicha tarea retiene el token y vuelve al filtro $\texttt{filter}_3$ con el siguiente subitem extraído de la cola `object_buffer` (`ndelayed`> 0); ver `(2)` en la figura 2.17). Este proceso se repite hasta que no quedan elementos pendientes en la cola (`ndelayed`=0), en cuyo caso, el token es liberado (ver `(3)` en la figura). De este modo, nuestra solución acaba con todos los subitems antiguos antes de procesar nuevos elementos. Esta solución garantiza la localidad en una estructura pipeline y limita el consumo de memoria en el buffer interno.

En la plantilla pipeline original, un filtro serie se ejecuta siempre por una única tarea en un momento dado. Además, si es un filtro ordenado (la opción por defecto, que es actualmente el caso para la etapa `SendBlock()`), tiene que procesar los elementos en orden, los tokens asociados son enumerados con un id ordenado (`id_token`). De este modo, cuando un token llega al filtro serie, no se procesa inmediatamente. Primero, la

maquinaria interna del filtro serie comprueba dentro de éste el próximo elemento en ser procesado. Esto está implementado a través de un `ordered_buffer` en el cual los tokens se insertan en sus posiciones correspondientes, donde esperan hasta su turno (figura 2.18(a)). La variable de buffer `low_token` mantiene el valor del token siguiente para procesarlo. Por tanto, si un token llega, y su `id_token` es más alto que `low_token`, entonces este token es guardado en `ordered_buffer`. Por contra, si el `id_token` es igual a `low_token`, el filtro serie procesa el elemento directamente y el valor de `low_token` se incrementa.




(a) (b)

Figura 2.18: Filtro serie: (a) implementación original; (b) nueva implementación.

Una vez que el filtro serie termina de procesar ese token, su maquinaria interna comprueba si hay un token esperando en la posición apuntada por `low_token`, en cuyo caso el elemento asociado es procesado por el filtro serie, el token es eliminado del buffer y el valor de `low_token` vuelve a incrementar.

Debido a la introducción de nuestro filtro multioutput, hemos tenido que cambiar la implementación del filtro serie. En este caso, los nuevos subitems generados por el filtro multioutput deben ser procesados en el orden en el que llegan al filtro serie (ordenado). El problema es que cuando un token con `id_token`$= i$ llega a un filtro multioutput, se generan nuevos subitems que deberían ser procesados en un filtro serie antes de los elementos asociados con otros tokens con `id_token` $> i$. Para lograr esto, asignamos a cada token un identificador más completo formado por una tupla <`id_token`, `id_subitem`>. Aquí, `id_token` es el id asociado al token inicial que entró en el filtro multioutput, mientras `id_subitem` es el id de cada subitem nuevo que el filtro multioutput genera para el token de entrada. En la figura 2.18(b) podemos ver que el buffer ordenado mantiene dos punteros: `low_token` para identificar el token actual y `low_subitem` para identificar el siguiente subitem esperando a ser procesado en el filtro serie. Como podemos ver en la figura, cada posición en el buffer

ordered_buffer no almacena los tokens que han llegado sino que hay un puntero a un nuevo ordered_buffer con el número de subitems generados por el filtro multioutput para cada id_token. Los tokens son guardados en esos buffers y cuando la tupla asociada a un token <id_token, id_subitem> es igual a <low_token, low_subitem> se retira el token. En particular, cuando un token con id=<low_token, low_subitem> llega al filtro serie, es directamente procesado. En el siguiente paso, <low_token, low_subitem> debe ser incrementado para señalar al siguiente token para ser procesado. Si low_subitem no ha alcanzado la última posición correspondiente al buffer, entonces es incrementado. En otro caso, se incrementa low_token y low_subitem es actualizado a uno.

La figura 2.19 muestra un fragmento de código de la implementación de dedup basada en el filtro multioutput, donde las líneas 2-6 muestran la definición de lo filtros: la línea 3 define el filtro multioutput, las líneas 21-25 construyen el pipeline TBB y la línea 20 ejecuta el pipeline. A esta implementación la denominamos implementación Multioutput.

### 2.5.3. Discusión y evaluación de las implementaciones

Podemos examinar varios aspectos en las tres implementaciones de dedup descritas anteriormente. Por ejemplo, desde el punto de vista del programador, la productividad es una meta importante cuando la codificación no es trivial para pipelines paralelos. Según esta perspectiva, los fragmentos de código mostrados en las figuras 2.13, 2.14 y 2.19 nos proporcionan algunos datos interesantes sobre la complejidad de programación de cada uno de ellos: la implementación híbrida TBB+Pthreads es la más compleja de implementar y de analizar, seguida de la versión Anidada y de la implementación Multioutput que sería la más simple. Por ejemplo, el archivo encoder.c, que es donde el pipeline se actualiza y ejecuta, necesita 759, 444 y 403 SLOC (número de líneas de código) para la versión Híbrida, la versión Anidada y la versión Multioutput respectivamente. Esta reducción importante del número de líneas de código de las versiones de TBB se debe principalmente a que éstas evitan la gestión de la cola de Pthread y el reordenamiento de los subitems en el chunk, que en TBB se realiza automáticamente mediante la maquinaria interna del filtro serie. Las líneas de código que ahorra la versión Multioutput respecto a la versión Anidada se deben a la declaración del pipeline anidado así como al filtro ReassembleBlocks(). Esta funcionalidad es desarrollada internamente por una implementación nueva del filtro en serie, para la cual hemos añadido soporte para controlar los nuevos elementos generados por el filtro multioutput cuando llegan a la etapa serie, como explicamos en la sección anterior.

```
1 // Pipeline TBB classes
2 class  DataProcess: tbb::filter {...};
3 class  FindAllAnchors:tbb::filter {.../*multioutput*/true ...};
4 class  ChunkProcess:tbb::filter {...};
5 class  Compress:tbb::filter {...};
6 class  SendBlock:tbb::filter {...};
7
8 ....
9 main() {
10   // Start task scheduler
11  tbb::task_scheduler_init init (nthreads);
12
13  tbb::pipeline pipeline;
14
15  Dataprocess dataprocess (inputfile);
16  FindAllAnchors findallanchors;
17  ChunkProcess chunkprocess;
18  Compress compress;
19  SendBlock sendblock (outputfile);
20
21  pipeline.add_filter(dataprocess);
22  pipeline.add_filter(findallanchors);
23  pipeline.add_filter(chunkprocess);
24  pipeline.add_filter(compress);
25  pipeline.add_filter(sendblock);
26
27 // Run the TBB pipeline
28  pipeline.run(ntokens);
29
30 // Clear the pipeline
31  pipeline.clear ();
32   ...
33 };
```

Figura 2.19: Uso de la plantilla pipeline con filtro multioutput en la nueva implementación TBB de `dedup`.

Otra cuestión que puede ser analizada es relativa al planificador de tareas y a la gestión del overhead para cada implementación. En las tres implementaciones, el framework de TBB gestiona el lanzamiento de tareas, el reciclado de tareas cuando un elemento es pasado a un nuevo filtro y el robo de tareas cuando un thread está ocioso. Sin embargo, hay overheads adicionales en la versión Anidada: i) cada vez que un token (asumimos que hay $ntokens$ en nuestro pipeline) alcanza el pipeline anidado, un nuevo pipeline es creado y destruido, y ii) cada vez que un pipeline es creado, su primer filtro (`FindAllAnchors`()) lanza $ntokens\_in$ nuevas tareas (asumiendo que el pipeline interno ha sido configurado con el parámetro $ntokens\_in$, como muestra la figura 2.14)

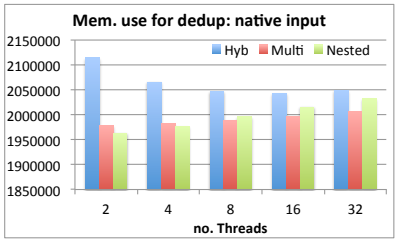
para procesar los subitems asociados con ese token.

El número total de tareas lanzadas en la ejecución del programa será $nitems + nitems \cdot nsubitems$, siendo $nitems$ el número total de elementos procesados en el programa y $nsubitems$ la media del número de subitems generados por la etapa multioutput. Esto puede ser una importante fuente de overhead, especialmente si la granuralidad del trabajo desarrollado por los filtros internos es fino [26]. Sin embargo, en la versión Híbrida y la implementación Multioutput, el pipeline es creado y destruido una única vez y hay sólo $ntokens$ lanzados en la ejecución total del programa.

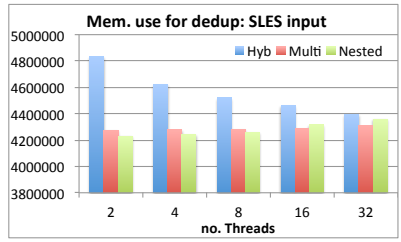
Otro tema es concerniente a la memoria consumida por la maquinaria interna de la implementación. En la implementación Híbrida, el tamaño total de las $nthreads - 1$ colas de tareas internas es $ntokens$ y el máximo tamaño de la cola externa para enviar los elementos a la etapa de salida es $ntokens \cdot nsubitems$. Las $nthreads$ colas de tareas en la versión Anidada tiene un tamaño total de $ntokens \cdot ntokens\_in$ elementos. Además, esta implementación también crea más objetos temporales en memoria, debido a la definición y creación de un objeto por cada filtro anidado cada vez que un token alcanza el pipeline anidado: en `dedup`, $5 \cdot ntokens + 3$ es el número de filtros reservados en memoria para el pipeline externo, mientras el número total de filtros reservados en la ejecución completa es $5 \cdot nitems + 3$. En las implementaciones Híbrida y Multioutput sólo sea crea un objeto por filtro durante la ejecución completa del programa (4 en la versión Híbrida y 5 en la versión Multioutput). Mirando la implementación Multioutput, el tamaño total de las $nthreads$ colas de tarea internas es $ntokens$, y el tamaño máximo del buffer `delayed` interno es $ndelayed = ntokens \cdot nsubitems$. Además, desde el punto de vista del consumo de memoria, podemos ver que en las tres implementaciones el consumo de los recursos está acotado. A continuación realizamos una evaluación del uso de memoria para cada implementación de `dedup` utilizando el comando `memusg` [75]. Nuestra plataforma es una máquina con cuatro socket de 8 cores Intel Xeon® CPU X7550 2.00GHz (32 cores) con sistema operativo SUSE 11.1. Los códigos fueron compilados con icc 11.1 y TBB 4.1. Se ejecutaron 5 veces, para luego calcular los valores medios. La Fig. 2.20 representa el uso de memoria en nuestra plataforma para tres conjuntos de entrada distintos, native (el fichero de evaluación original de PARSEC 2.1 de tamaño 672 MB); SLES (el fichero ISO del SLES11 de tamaño 2.7GB) y XML (un fichero XML extraído de la wikipedia de tamaño 6.74 GB). `Hyb`, `Multi` y `Nested` representa los picos de memoria usada en bytes para la versión Híbrida, Multioutput y Anidada respectivamente.

La implementación Híbrida consume claramente más memoria, aunque su uso de memoria tiende a decrecer conforme aumentamos el número de threads para todos los tamaños de entrada. La pila de procesos Pthreads, así como la cola externa utilizada para enviar los subitems a la etapa de salida, son responsables de las diferencias entre el uso de memoria de esta implementación y las otras dos. Por otro lado, las implemen-

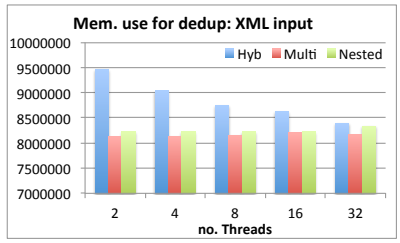
taciones Multioutput y Anidada tienen un requerimiento similar de memoria, aunque la implementación Anidada parece guardar más objetos temporales, especialmente cuando el número de threads crece, lo cual explica porque hay más memoria utilizada. Este incremento del consumo de memoria en la implementación Anidada está relacionado con la degradación de rendimiento que ocurre cuando se alcanza el cuello de botella de Entrada/Salida.




(a) entrada native




(b) entrada SLES

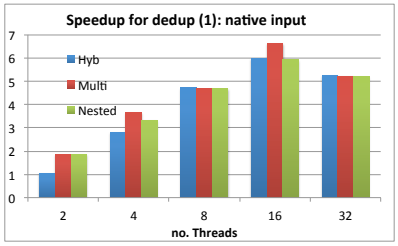



(c) entrada XML

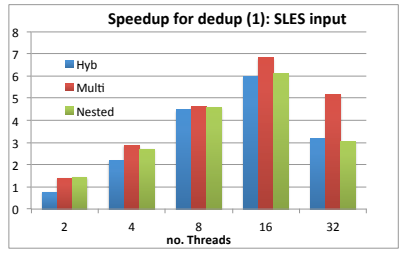
Figura 2.20: Memoria usada (Bytes) para las tres implementaciones de `dedup`.

Veamos el rendimiento de cada implementación. Para hacer eso, medimos el speedup de cada una de ellas en nuestra plataforma. El speedup es computado respecto al tiempo secuencial de `dedup` (PARSEC proporciona una versión secuencial). La figura 2.21 representa el speedup para los tres conjuntos de entrada que mostramos anteriormente: native, SLES y XML. `Hyb`, `Multi` y `Nested` representan los valores para las tres implementaciones respectivamente. Los resultados muestran que las implementaciones escalan hasta 16 threads, cuando se alcanza el cuello de botella de Entrada/Salida en las etapas serie. La iplementación Multioutput siempre mejora a las otras dos versiones, especialmente cuando el número de threads se incrementa y el tamaño de entrada del problema es más grande. Este rendimiento se explica por una mejor utilización de los recursos, menos overhead y mejor comportamiento de localidad. La implementación Híbrida ofrece peores resultados, particularmente cuando el número de threads es pequeño. En este caso, como hemos estudiado, el principal factor es la pobre utilización

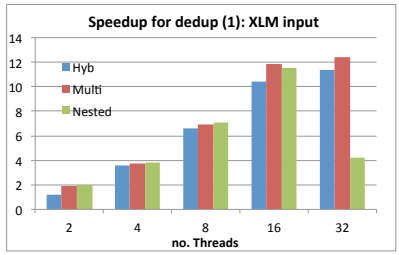
de los recursos. El rendimiento de la versión Anidada se sitúa entre la Híbrida y la Multioutput. En este caso, la implementación Anidada tiene un overhead adicional relativo a la creación y destrucción de los pipeline internos y al coste del lanzamiento de más tareas dentro del pipeline interno, lo cual explica porque el rendimiento de esta implementación es peor que la implementación Multioutput. De hecho, cuando se alcanza el cuello de botella y el tamaño de entrada es más grande que el native (las entradas SLES y XML), como podemos ver, la versión Anidada sufre una degradación significante.




(a) entrada native




(b) entrada SLES




(c) entrada XML

Figura 2.21: Speedup para las tres implementaciones de `dedup`: a) entrada native; b) entrada SLES; y c) entrada XML.

Esto es debido a un sutil detalle que diferencia el comportamiento entre la Multioutput y la versión Anidada. Cuando un pipeline anidado es creado para un token de un pipeline externo, la ejecución del thread procesará $nsubitems$ de estos pipeline internos. Cada vez que un token es liberado en el último filtro del pipeline interno, se lanza una nueva tarea y empieza a ejecutarse el primer filtro en la que el pipeline interno procesa un nuevo subitem. El efecto es que cada vez, un pipeline interno procesa todos los subitems que un elemento del pipeline externo ha creado. En relación con el pipeline externo, podría haber elementos viejos (chunks) en el filtro de salida en serie que no han sido finalizados todavía, mientras los elementos más nuevos ya han finalizado y llegado

al filtro de salida. Esto significa que esos elementos más jóvenes (fuera de orden) tiene que esperar a los elementos previos para llegar a la cola (debido al orden natural de la etapa de salida). Más específicamente, cuando un elemento fuera de orden llega a la etapa de salida, tiene que esperar su turno para ejecutar el último filtro `SendBlock`(). Recordamos que en la etapa paralela `ChunkProcess`(), una entrada nueva es insertada en una tabla hash para cada bloque único. La tabla hash sigue una estrategia de encadenamiento para evitar colisiones y las entradas nuevas se insertan al principio de la lista enlazada mientras un elemento espera para entrar en `SendBlock`(). Cuanto más tiempo espera un elemento, más tiempo se necesita para atravesar la lista enlazada para encontrar la entrada correspondiente cuando se ejecuta el filtro `SendBlock`(). Para esta versión Anidada en 32 cores, hemos medido incrementos de 2x a 10x en el número de operaciones de recorrido de la lista, dependiendo de la entrada del problema y del orden no determinístico en el que se ejecutan las tareas. No solo el recorrido de la lista consume tiempo, sino que también tiene un impacto fuerte sobre el porcentaje de fallos de caché como se muestra en la sección 2.7.1. Este efecto se nota más cuando el número de threads se incrementa en la etapa de salida serie. Nos referiremos de aquí en adelante a este efecto como *ineficiencia del drenado en serie* de la etapa de salida.

Sin embargo, en la versión Multioutput, cada vez que un token deja el último filtro, el procesamiento de la tarea irá al buffer `delayed` para obtener y procesar el primer subitem en la cola. Esto significa, que nuestra implementación Multioutput termina primero todos los subitems antiguos, los cuales pueden ayudar a completar los elementos más antiguos pendientes en la etapa de salida serie y además a drenar la etapa de salida más rápido. De este modo, el drenaje ineficiente de elementos de la etapa de salida serie en la implementacion Anidada es el factor principal que explica la degradación de esta implementación en 32 threads para las entradas de SLES y XML. Este factor es también relativo al uso de memoria en la implementacion Anidada cuando el número de threads se incrementa: el tamaño de la cola asociada con la salida del filtro en serie debe ser largo para mantener los elementos pendientes. Hay otra cuestión importante para saber si las unidades de procesamiento están bien o mal utilizadas. La versión Anidada y la implementación Multioutput promueven un paralelismo de grano fino, que junto con la estrategia work-stealing del planificador de TBB que lleva a cabo el balanceo de carga si fuera necesario, pueden garantizar la utilización completa de las unidades de procesamiento. Por contra, la implementación Híbrida se centra en paralelismo de grano grueso. Tiene dos pools de threads dedicados para diferentes partes del pipeline ($nthreads - 1$ para la parte de TBB y 1 para la parte de Pthread) que pueden comportarse de manera diferente.

## 2.6. Modelo analítico del pipeline con etapa multioutput

En esta sección presentamos nuevos modelos analíticos para la evaluación del rendimiento de las implementación pipeline descritas en la sección anterior. Más precisamente nos referimos a las implementaciones no triviales basadas en tareas para estructuras pipeline paralelas en las cuales uno de los filtros puede tener más tokens de salida que de entrada, como en el código `dedup`.

Una estructura general del grafo en este tipo de pipeline se muestra en la figura 2.22, donde cada token de entrada es procesado a través de filtros estándar ($F_1$, $F_2$, ..., $F_{m-1}$) que devuelven un token de salida hasta que se alcanza el filtro ($F_m$) donde se crean elementos nuevos a partir del token. En nuestros modelos, asumimos que el filtro $F_m$ produce $nsubitems$ subitems por cada token de entrada como se muestra en la figura. Todos los subitems nuevos pueden ser procesados independientemente entre el filtro $F_m$ y el último $F_n$.

En la figura, el último filtro del pipeline devuelve un token de salida por cada $nsubitems$ subitems de entrada, aunque podría devolver uno de salida por cada subitem de entrada. Esta situación también puede ser considerada en nuestro modelo. Además, en cada filtro $F_i$, $T_{ser_i}$ representa el tiempo de servicio para procesar un token en un filtro. En el caso de un filtro con Entrada/Salida múltiple de subitems, el tiempo de servicio que se muestra en la figura es el tiempo empleado para procesar todos los correspondientes subitems ($nsubitems$) en el filtro.

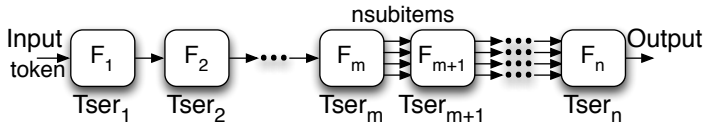



Figura 2.22: Estructura de un pipeline con filtro multioutput.

Recordemos que $ntokens$ es un parámetro de entrada que el programador proporciona al runtime de TBB para definir el número máximo de tareas que pueden existir simultáneamente en el sistema, mientras $nsubitems$ representa el número de subitems que dejan el filtro $F_m$. Para seleccionar $ntokens$, seguimos la metodología explicada en [57]. Esto nos ayuda a configurar este parámetro de manera que sea suficientemente grande como para mantener todos los threads en el pipeline TBB ocupados y además aseguren el máximo throughput efectivo, pero no demasiado grande para evitar el overhead debido a la gestión de tareas, sincronización y contención de las colas internas de las tareas. Por otro lado, el parámetro $nsubitems$ depende del problema.

Presentamos algunas características de las implementaciones estudiadas en detalle.

Caso 1: Implementación Híbrida TBB+Pthreads. Esta implementación utilizada como base es una de las implementaciones de tareas que permite filtros estándar de TBB y está descrita en la sección 2.5.1. Recordemos que esos filtros estándar devuelven un solo elemento por token de entrada, como describimos en la figura 1.10. Además, si un filtro produce más de un elemento de salida, entonces ese filtro y los siguientes deberían colapsarse en un filtro nuevo simple que internamente procese todos los subitems necesarios. La figura 2.23 muestra la estructura de este pipeline, donde $F_1$, $F_2$, ..., $F_{m-1}$ representan los filtros estándar de TBB y los filtros $F_m + ... + F_{n-1}$ representan el colapsado. Este filtro colapsado puede dejar los subitems en un buffer externo temporal antes de terminar. Solo necesitamos añadir la etapa de salida, $F_n$, en nuestro ejemplo, que es responsable de asegurar que los subitems encolados en el buffer sean procesados para salir correctamente. Esta nueva etapa, puede ser implementada utilizando Pthreads. Otra vez, $T_{ser_i}$ representa el tiempo de servicio para procesar n tokens en un filtro estándar, mientras $\sum_{j=m}^{n-1} T_{ser_j}$ representa el tiempo de servicio para procesar todos los subitems generados por un token, es decir, $nsubitems$, en el filtro colapsado. $T_{ser_n}$ es el tiempo de servicio para procesar $nsubitems$ en la etapa de salida. En el caso que la etapa de salida procese y devuelva cada subitem individualmente, el tiempo de servicio de este último sería $T_{ser_n}/nsubitems$.

Caso 2: Implementación de pipeline TBB basada en nuestro filtro multioutput. La implementación de tarea de este caso ha sido descrita en la sección 2.5.2 y también se puede ver en la figura 2.17. La figura 2.24 muestra la estructura de nuestro pipeline, donde $F_1$, $F_2$, ...,$F_{m-1}$, $F_{m+1}$,..., $F_n$ representan los filtros estándar de TBB y $F_m$ representa nuestro filtro multioutput, para el cual un buffer interno llamado `delayed` mantiene los $nsubitems$ subitems generados. $T_{ser_i}$, representa el tiempo para procesar un token en un filtro estándar anterior al filtro multioutput, mientras $T_{ser_i}/nsubitems\ i \in [m, n]$ representa el tiempo tomado para procesar un subitem desde el filtro multiotput hasta el filtro de salida.

Caso 3: Pipeline TBB basado en pipelines anidados. Esta implementación fue propuesta en [70]. La implentación de este caso ha sido descrita en la sección 2.5.1. La figura 2.25 muestra la estructura de nuestro pipeline, donde $F_1$, $F_2$, ..., $F_{m-1}$, $F_{m+1}$ son los filtros estándar, $F_{process}$ representa los filtros internos encapsulados creados para procesar los subitems generados en la etapa multioutput $F_m$. $T_{ser_i}$, representa el tiempo para procesar un elemento en un filtro estándar en el pipeline externo, y $T_{ser_i}/nsubitems$ representa el tiempo tomado para procesar un subitem en uno de los pipeline internos. Como se explica en [70], la etapa de salida original se descompone en dos filtros: $F_{na}$ controla el reordenamiento de

los subitems antes de pasar los tokens al pipeline externo; y $F_{nb}$ es el filtro del pipeline externo que controla la salida. $T_{ser_{na}}/nsubitems$ representa el tiempo tomado para procesar un subitem en $F_{na}$ y $T_{ser_{nb}}$ el tiempo tomado para procesar un elemento en $F_{nb}$. Asumimos que $T_{ser_{na}} + T_{ser_{nb}}$ es equivalente para $T_{ser_n}$ en la etapa de salida original.

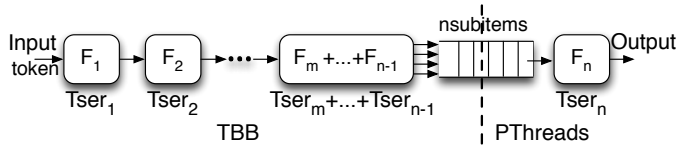



Figura 2.23: Estructura del pipeline para el caso 1: versión Híbrida (TBB + Pthreads).

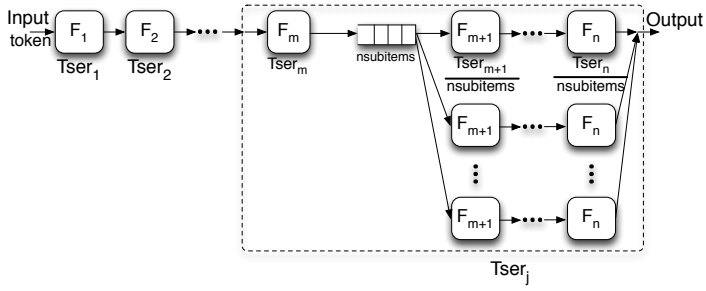



Figura 2.24: Estructura del pipeline para el caso 2: TBB basado en filtro multioutput.

El objetivo de nuestro modelo es comprender mejor las relaciones entre los recursos y los compromisos de rendimiento implicados en cada implementación. En nuestros modelos, no consideramos los overheads relativos a colas, threads o gestión de tareas, aunque algunos de esos overheads serán medidos en la sección 2.7. Por tanto, los modelos deberían ser interpretados como límites optimistas del comportamiento de cada implementación. Como una primera métrica de rendimiento, con objeto de comprobar la validez de nuestros modelos, pondremos la atención en la estimación de los tiempos de ejecución para las diferentes implementaciones.

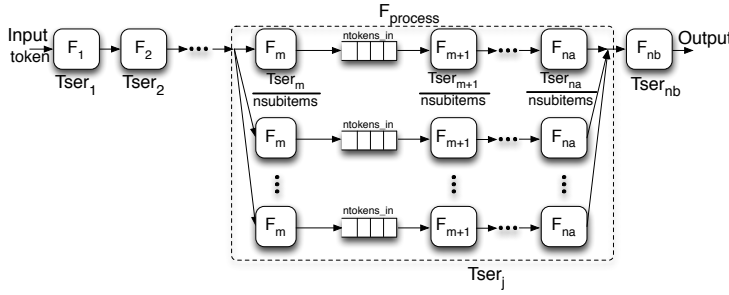



Figura 2.25: Estructura del pipeline el caso 3: TBB basado en pipelines anidados.

| **Parámetro** | **Definición** | **Valor** |
|---|---|---|
| $nthreads$ | Número de threads paralelos | $nthreads = 2 : 32$ |
| $ntokens$ | Capacidad del sistema TBB | Parámetro de entrada |
| $nsubitems$ | Capacidad del buffer para el filtro multioutput | Parámetro del problema |
| $nitems$ | Número de items para ser procesados | Tamaño de entrada |
| $T_{ser}$ | Tiempo de servicio (tiempo para procesar un item) | Distribución exponencial |
| $T_{arr}$ | Tiempo de llegada de un items | Distribución exponencial |
| $c$ | Número de threads por cola lógica | Caso 1: TBB $c = nthreads - 1$ \| Pth $c = 1$<br>Casos 2, 3: $c = nthreads \mid c' = 1$ |
| $N$ | Capacidad del buffer del sistema | Caso 1: TBB $N = ntokens$ \| Pth $N = \infty$<br>Caso 2: $N = ntokens \mid N' = ntokens \cdot nsubitems$<br>Caso 3: $N = ntokens \mid N' = ntokens \cdot ntokens\_in$ |
| $K$ | Población del sistema | $K = N$ |
| *Métrica de rendimiento* | | |
| $\lambda_e$ | Throughput efectivo | Caso 1: sección 2.6.1<br>Caso 2: sección 2.6.2<br>Caso 3: sección 2.6.3 |
| $Time$ | Tiempo de ejecución estimado | Caso 1: eq. 2.2<br>Caso 2: eq. 2.16<br>Caso 3: eq. 2.18 |

Tabla 2.3: Parámetros del modelo analítico.

Los parámetros de los modelos se resumen en la tabla 2.3. En nuestros modelos, utilizaremos el concepto de cola lógica o sistema de colas para representar: i) el buffer donde los elementos pendientes esperan para ser procesados (dependiendo del caso, una

cola guarda tokens o subitems); y ii) los servidores (o threads) que procesan los elementos. Todas las implementaciones pueden ser modeladas como un sistema cerrado [57], para el cuál el sistema de colas también incluye una representación de la población que puede solicitar un servicio.

En las siguientes secciones explicaremos cada caso en más detalle.

### 2.6.1. Caso 1. Pipeline Híbrido TBB y Pthreads

En esta implementación del pipeline tenemos que modelar el comportamiento de dos sistemas de colas que están conectados a través de un buffer externo. Por tanto, pueden ser modelados como una red de dos colas lógicas [3]. Los modelos se muestran en la figura 2.26.

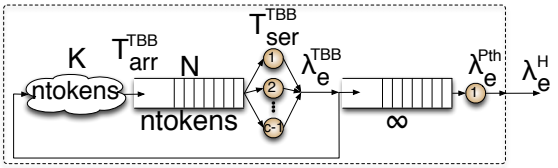



Figura 2.26: Modelo para el caso 1: pipeline de TBB y Pthreads.

Las dos colas lógicas modelan la siguiente restricción: los servidores del pool (threads) que trabajan para el pipeline TBB no pueden participar en el procesado de elementos de la parte de Pthreads, y viceversa. En otras palabras, los threads de cada cola lógica están exclusivamente dedicados a esa cola. Ambas colas trabajan concurrentemente y el flujo de elementos entre ellas tiene diferentes throughputs efectivos:

- $\lambda_e^{TBB}$, el throughput efectivo para los elementos que dejan el pipeline (salida del filtro colapsado),

  y

- $\lambda_e^{Pth}$, el throughput efectivo para los elementos que dejan la última etapa $F_n$.

En una red, el throughput efectivo del sistema completo vendrá dado por la etapa más lenta. Esto es, el throughput efectivo en nuestra implementación Híbrida viene dado por la eq. 2.1, mientras el tiempo estimado de ejecución de nuestro código depende de este throughput efectivo para procesar un elemento ($\lambda_e^H$) y el número total de elementos ($nitems$) procesados, como muestra la eq. 2.2.

$$\lambda_e^H = min(\lambda_e^{TBB}, \lambda_e^{Pth}) \quad (2.1)$$

$$Time^H = \frac{nitems}{\lambda_e^H} \quad (2.2)$$

Ahora mostramos como calcular el throughput efectivo para cada cola lógica.

- La primera cola lógica representa la implementación TBB basada en un pipeline de filtros estándar $F_1$, $F_2$,..., $F_{m-1}$ y los colapsados $F_m$+...+$F_{n-1}$. Estudiamos el pipeline TBB basado en filtros estándar en [57], encontrando que idealmente son sistemas cerrados que pueden ser modelados como una cola $M/M/c/N/K$ [3]. En notación de colas estándar, esto representa una cola lógica con tiempos de llegada y servicio que siguen una distribución exponencial ($M$, $M$). Esto es debido a que asumimos que los tiempos de llegada y servicio de elementos siguen un proceso de Poisson. Esta suposición no compromete la robustez de nuestro modelo, porque se pueden utilizar distribuciones alternativas para tiempos de servicio y llegada, como por ejemplo una función o un simulador para medir las cantidades medias. Adicionalmente, $c$ representa el número de servidores (threads trabajadores), $N$ es el tamaño del sistema (el tamaño de la cola para los elementos pendientes) y $K$ es la población del sistema [68]. En este estudio, asumimos que la implementación de TBB del pipeline se comporta como una cola global con $c = nthreads - 1$ threads. Esto representa un caso asintótico optimista donde la característica work-stealing de TBB balancea el trabajo perfectamente entre los threads. Los elementos son las tareas del sistema, y en el régimen permanente, hay a lo sumo $ntokens$ tareas en el sistema. Esas tareas representan la población y también representan la capacidad de las colas globales, esto es $K = N = ntokens$.

  En el régimen permanente, la implementación de TBB puede ser vista como un sistema global como hemos mencionado. Esto es debido a que una tarea nueva (token) no puede entrar hasta que una previa ha salido del sistema. Esto es como si los tokens continuasen circulando sin dejar nunca las colas. En este caso, cuando un token termina, devuelve todos los subitems producidos y los encola en la salida del filtro colapsado. Por tanto, cada tarea lleva a cabo el trabajo desde el filtro $F_1$ a la salida del filtro colapsado a través de filtros intermedios del pipeline (ver figura 2.26). Así, el tiempo de servicio medio para esta cola, $T_{ser}^{TBB}$, depende del tiempo que necesita un token para salir del filtro colapsado,

$$T_{ser}^{TBB} = \sum_{i=1}^{n-1} T_{ser_i} \quad (2.3)$$

Una observación importante es que el tiempo medio para la llegada de un token, $T_{arr}^{TBB}$, también depende del procesado de un token, así que,

$$T_{arr}^{TBB} = T_{ser}^{TBB} \tag{2.4}$$

En nuestro modelo de colas, la ecuación 2.7 nos da $\lambda_e^{TBB}$. Esto es el throughput efectivo para el token que deja el filtro colapsado. Depende de $T_{ser}^{TBB}$, $T_{arr}^{TBB}$ y $P_k$. Este último parámetro es la probabilidad de que haya $k$ tokens en las colas del sistema de TBB (eqs. 2.5 - 2.6, ver [3]). $P_0$ representa la probabilidad de que no haya tokens en la cola.

$$P_0 = \left\{ \sum_{k=0}^{c-1} \binom{N}{k} \cdot \left( \frac{T_{ser}^{TBB}}{T_{arr}^{TBB}} \right)^k + \sum_{k=c}^{N} \frac{N!}{(N-k)! \cdot c! \cdot c^{k-c}} \cdot \left( \frac{T_{ser}^{TBB}}{T_{arr}^{TBB}} \right)^k \right\}^{-1} \tag{2.5}$$

$$P_k = \begin{cases} \binom{N}{k} \cdot \left( \frac{T_{ser}^{TBB}}{T_{arr}^{TBB}} \right)^k \cdot P_0 & k = 0, 1, \ldots, c-1 \\ \frac{N!}{(N-k)! \cdot c! \cdot c^{k-c}} \cdot \left( \frac{T_{ser}^{TBB}}{T_{arr}^{TBB}} \right)^k & k = c, c+1, \ldots, N \end{cases} \tag{2.6}$$

$$\lambda_e^{TBB} = \sum_{n=0}^{N} (N-n) \cdot \frac{P_n}{T_{arr}^{TBB}} \tag{2.7}$$

- La segunda cola lógica representa la implementación Pthreads de la última parte del pipeline, que es la parte que se encarga de los múltiples subitems del filtros colapsado. En este caso, representa el procesamiento de la cola de subitems en el buffer externo a través de la etapa de salida. El tamaño del buffer externo es aproximadamente $ntokens \cdot nsubitems$, el cual se asume que es suficientemente grande como para que la cola tenga capacidad infinita ($N = \infty$). Cuando la capacidad es infinita, y el régimen permanente, esta cola puede ser vista como un sistema abierto. Asumiendo los tiempos de llegada y servicio de tipo exponencial, este segundo sistema se comporta como una cola $M/M/c$, con $c = 1$. Hemos dedicado solo un thread para este filtro, porque hemos asumido que esta etapa de salida procesará el token en serie, como pasa en `dedup` y en otras aplicaciones de pipeline [57]. Si queremos modelar una etapa paralela, podríamos dedicar más de un thread y seleccionar $c = d > 1$, pero entonces el número de threads dedicados en la cola lógica de TBB sería $nthreads - d$. En cualquier caso, en esta segunda cola, el tiempo medio de servicio, $T_{ser}^{Pth}$, depende del procesado (por etapa $F_n$) de todos los subitems que un elemento (token) deja en el buffer externo.

$$T_{ser}^{Pth} = T_{ser_n} \tag{2.8}$$

Como mencionamos, en el caso que esta etapa procese y devuelva cada subitem individualmente, el tiempo de servicio sería $T_{ser_n}/nsubitems$. El tiempo medio de llegada de un elemento a esta cola, $T_{arr}^{Pth}$, depende ahora del throughput efectivo desde la cola previa,

$$T_{arr}^{Pth} = \frac{1}{\lambda_e^{TBB}} \tag{2.9}$$

En este modelo, la eq. 2.10 nos da $\lambda_e^{Pth}$, el cual es el throughput efectivo para los elementos que dejan la etapa de salida $F_n$,

$$\lambda_e^{Pth} = \frac{1}{T_{ser}^{Pth}} \tag{2.10}$$

### 2.6.2. Caso 2. Pipeline TBB basado en el filtro multioutput

En esta implementación de pipeline, podemos modelar nuestro sistema como una cola cerrada de tipo $M/M/c/N/K$, como se muestra en la figura figura 2.27.

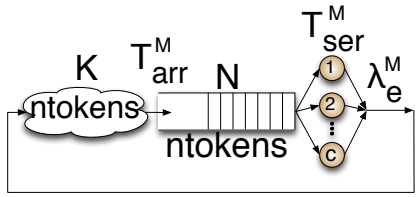



Figura 2.27: Modelo para el caso 2: pipeline TBB basado en el filtro multioutput.

Nuestro pipeline de TBB se comporta como una cola global con $c = nthreads$, asume de nuevo que a lo sumo hay a $ntokens$ tareas en el sistema, por tanto, la población y capacidad de la cola global vienen dadas por $K = N = ntokens$. Aunque presentamos nuestro sistema como de tipo cerrado, una diferencia importante respecto al modelo híbrido es que con un filtro multioutput, un token es liberado después de que todos los subitems almacenados en el buffer interno `delayed` han sido procesados completando el último filtro $F_n$ (es decir $ndelayed = 0$) (ver figura 2.17). Esto significa que todos los subitems pueden ser procesados concurrentemente por $nthreads$, en vez de ser procesado por un thread, como ocurre en el sistema híbrido. Debido a esto, en el sistema el tiempo medio de servicio para procesar un token , $T_{ser}^{M}$ (y también la media de los tiempos de llegada, $T_{arr}^{M}$) tiene dos componentes,

$$T_{ser}^{M} = T_{arr}^{M} = \left( \sum_{i=1}^{m-1} T_{ser_i} \right) + T_{ser_j} \tag{2.11}$$

La primera componente (el sumatorio) representa el procesamiento de un token hasta que entra en el filtro multioutput ($F_m$), mientras $T_{ser_j}$ representa el tiempo medio de servicio efectivo que el filtro multioutput y los siguientes filtros necesitan para procesar los subitems guardados en el buffer interno. Idealmente, el filtro $F_m$ puede ser visto como un procesado de subitems individuales. Recordemos la figura 2.17, en la que cada vez que una tarea con un token alcanza la última etapa en un pipeline con un filtro multioutput, en vez de liberar un token, éste comprueba si hay todavía subitems esperando ser procesados en el buffer interno ($nsubitems > 0$). En este caso, el token es utilizado por la tarea para empezar en el filtro $F_{m+1}$ con un subitem sacado del buffer. Dado que alguna tarea (y el correspondiente thread) puede coger un subitem del buffer y que todos los threads colaboran en el procesado de los subitems, podemos ver esta parte del pipeline como un grupo de threads acoplados que trabajan en paralelo donde cada thread es el servidor de una cola que almacena los subitems para ser procesados. Ocasionalmente, los otros threads remotos en el sistema pueden robar trabajo (subitems) de las colas locales. Cada vez que un subitem es procesado por un thread, entonces un slot pasa a estar disponible en esta cola local y un subitem nuevo puede entrar en el sistema. Debido a esto, cada thread puede ser visto como una cola en un modelo cerrado, esto es, cada thread puede ser modelado como una sistema de colas $M/M/c'/N'/K'$, el cual mostramos en la figura 2.28(a). Para esta cola, cada tarea lleva a cabo el trabajo de un subitem desde el filtro $F_m$ (incluimos el procesamiento de un subitem en el filtro multioutput) hasta $F_n$. Además, el tiempo medio de servicio, $T'_{ser}$, y el tiempo de llegada, $T'_{arr}$ en el sistema de colas $M/M/c'/N'/K'$ depende del procesamiento de un subitem y puede ser calculado como:

$$T'_{ser} = T'_{arr} = \sum_{i=m}^{n} \frac{T_{ser_i}}{nsubitems} \tag{2.12}$$

El número de servidores ($c'$) por cola es inicialmente un thread ($c' = 1$), porque queremos modelar el comportamiento de un thread. Esto es así ya que todos los threads trabajan en paralelo y este modelo nos da el comportamiento promedio de uno de ellos. La capacidad total del sistema es finita y viene definida por el número de subitems que podrían existir simultáneamente en el sistema: $ntokens \cdot nsubitems$. Inicialmente, asumimos que las colas lógicas tienen igual capacidad para todos los threads y que los subitems están equitativamente distribuidos, es decir $\forall\ i = 1 : nthreads, N' = ntokens \cdot nsubitems/nthreads$. Esto es, $N'$ representa el número máximo de subitems

por threads. La población que puede solicitar el servicio es finita e igual que la capacidad, es decir $K' = N'$.

En [57] demostramos que el concepto de work-stealing es similar al de compartición de trabajo (*load sharing*), y presentamos un modelo analítico para considerar el efecto en un pipeline TBB que procesase tokens en vez de subitems. La metodología presentada en el estudio es aplicable directamente a esta parte de nuestro pipeline, donde estamos interesados en modelar la compartición de subitems mediante work-stealing. A continuación resumimos este modelo para work-stealing basado en los tres pasos siguientes que fueron inicialmente introducidos en [77] y que ilustramos en la figura 2.28.

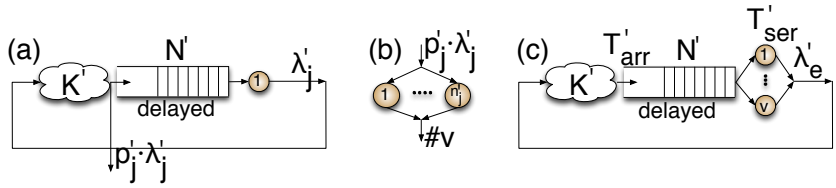



Figura 2.28: Modelo para work-stealing: (a) Paso 1: una cola de thread ; (b) Paso 2: robo de subitems; (c) Step 3: efecto de robo en la cola del thread.

**Paso 1.** Cada thread se comporta como una cola local $M/M/c'/N'/K'$ (ver figura 2.28(a)). Inicialmente $c' = 1$ y $N' = K' = ntokens \cdot nsubitems/nthreads$, como estado previo. Desde este paso obtenemos $\lambda'_j$ (el throughput interno para cada thread) y $p'_j$ (la probabilidad que el thread j-th esté ocupado [77]).

**Paso 2.** El robo de elementos para otros threads puede ser modelado como un sistema $M/M/n'_j/n'_j$. Para cada thread, hemos utilizado $p'_j \cdot \lambda'_j$ como carga ofrecida (ver figura 2.28(b)) para robar subitems y $n'_j$ es la capacidad remota disponible para desarrollar el robo. $n'_j$ se determina por el número y el factor de utilización [1] que potencialmente pueden robar, $\rho'_k$. Veamos como calcular $n'_j$. Un thread $k$ remoto que opera independientemente tiene, en media, un capacidad ociosa de $(1-\rho'_k)\cdot c'_k$. Dado que el factor de utilización de los threads remotos practicamente no se ve afectado por el work-stealing, un thread $j$ tiene a su disposición, en media, una capacidad de trabajo ociosa total de,

$$n'_j = nidle'_j = \sum_{\substack{k=1:nthreads \\ j \neq k}} (1 - \rho'_k) \cdot c'_k \tag{2.13}$$

[1] Este parámetro viene de teoría de colas y representa la probabilidad de que un servidor (o thread en nuestro modelo) esté ocupado.

Una vez calculado $n'_j$ para cada thread, modelamos ahora los subitems robados por los threads remotos mediante una cola $M/M/n'_j/n'_j$ . Este nuevo sistema de colas tiene una tasa de llegada $p'_j \cdot \lambda'_j$ y un tiempo de servicio $T'_{ser}$. A continuación, utilizando las ecuaciones para este modelo de colas [3], calculamos $v$, el cual es la media del número de servicios activos llevados a cabo en esta cola (es decir los subitems robados [77]).

**Paso 3.** Cuando incorporamos work-stealing en nuestro modelo, cada thread es modelado como un sistema de cola $M/M/c' + v/N'/K'$; es decir, cada thread se comporta como si tuviese a su disposición una capacidad de trabajo (Virtual), $v$, adicional (ver, figura 2.28(c)) así que ahora recalculamos el throughput interno $\lambda'_j$, teniendo en cuenta la nueva capacidad de trabajo.

El throughput efectivo por thread en este sistema, $\lambda'_e$, es determinado por la media de los throughputs internos que cada thread proporciona, esto es,

$$\lambda'_e = mean_{j=1}^{nthreads}(\lambda'_j) \tag{2.14}$$

Una vez hemos calculado el throughput efectivo que proporciona cada thread al procesar concurrentemente subitems ($\lambda'_e$), y dado que estamos interesados en calcular el tiempo de servicio que esta parte del pipeline emplea en computar el número de subitems asociados a un token de entrada ($nsubitems$), podemos calcular este tiempo como,

$$T_{ser_j} = \frac{nsubitems}{\lambda'_e} \tag{2.15}$$

Esta es la segunda componente de la eq. 2.11 y representa el tiempo medio de servicio efectivo que el filtro multioutput y los siguientes filtros necesitan en media para procesar, los $nsubitems$ por token. Utilizando esto, calculamos $T^M_{ser}$ (y $T^M_{arr}$, eq. 2.11) para nuestra cola global $M/M/c/N/K$ (Fig 2.27). Ahora, podemos utilizar la ecuación del modelo cerrado, más específicamente las eqs. 2.5, 2.6 y 2.7, pero reemplazando $T^{TBB}_{ser}$ y $T^{TBB}_{arr}$ por $T^M_{ser}$ y $T^M_{arr}$, para calcular $\lambda^M_e$, el cual es el throughput efectivo para procesar un elemento en nuestra cola global.

De nuevo, el tiempo estimado de ejecución de nuestra implementación dependerá del throughput efectivo para procesar un elemento ($\lambda^M_e$) y el número total de elementos ($nitems$) procesados, como vemos en la figura eq. 2.16.

$$Time^M = \frac{nitems}{\lambda^M_e} \tag{2.16}$$

### 2.6.3. Caso 3. Pipeline TBB basado en pipelines anidados

En esta implementación de pipeline podemos modelar nuestro sistema como un sistema global cerrado basado en una cola $M/M/c/N/K$. Esto es, el pipeline TBB externo se comporta como una cola global con $c = nthreads$ threads, y asumiendo otra vez que hay a lo sumo $ntokens$ tareas en el sistema, entonces la población y la capacidad de la cola global viene dada por $K = N = ntokens$.

Cada token que alcanza la etapa multioutput crea un pipeline anidado con $ntokens\_in$ subitems que pueden ser procesados concurrentemente por $nthreads$. De este modo, en el sistema, el tiempo medio de servicio para procesar un token $T^N_{ser}$ (y también medio de llegada, $T^N_{arr}$) tiene dos componentes.

$$T^N_{ser} = T^N_{arr} = \left( \sum_{i \ in \ outer \ pipe} T_{ser_i} \right) + T_{ser_j} \tag{2.17}$$

El primer componente (la suma) representa el procesamiento de un elemento en el pipeline externo, mientras $T_{ser_j}$ representa el tiempo medio efectivo que un pipeline anidado necesita para procesar los subitems generados por un token desde el pipeline externo. En este modelo, cada vez que una tarea con un token (subitems) alcanza el último filtro en un pipeline interno ($F_{na}$), el subtoken se libera, volviendo al primer filtro del pipeline interno ($F_m$) para obtener un nuevo subitem de su cola hasta que todos los subitems generados por el token externo que hayan sido procesados. Cuando un thread está ocioso, puede robar trabajo de otra cola remota y colaborar en el procesamiento de subitems. De este modo, como en el caso 2, podemos ver esta parte del pipeline como un grupo de threads acoplados en paralelo en el cual cada thread es el servidor de una cola local en la que se almacenan los subitems que van a ser procesados. Ocasionalmente, los otros threads remotos en el sistema pueden robar trabajo (subitems) de la cola local. De este modo, cada thread puede ser visto como una cola en un modelo cerrado: esto es, cada thread puede ser modelado otra vez como un sistema $M/M/c'/N'/K'$ de cola, similar a la que muestra la figura 2.28. Cada tarea lleva a cabo el trabajo de un subitem desde el filtro $F_m$ hasta $F_{na}$. Por tanto, el tiempo medio de servicio $T'_{ser}$, y el tiempo de llegada, $T'_{arr}$ ahora dependen del procesamiento de un subitem, y pueden ser calculados mediante la ecuación 2.12.

Al igual que en el caso 2, el número de servidores ($c'$) por cola es inicialmente un thread ($c' = 1$) porque queremos modelar el comportamiento de un thread. La capacidad total del sistema es finita y viene definida por el número de subitems que podrían existir simultáneamente en el sistema: $ntokens \cdot ntokens\_in$. Inicialmente asumimos que las colas lógicas para todos los threads tienen igual capacidad y que los

subitems están equitativamente distribuidos. Es decir, $\forall\ i = 1 : nthreads, N' = ntokens \cdot ntokens\_in/nthreads$. Esto es, $N'$ representa el número máximo de subitems por thread. La población que puede solicitar el servicio es finita e igual que la capacidad, que es $K' = N'$.

Podemos seguir los mismos tres pasos del procedimiento descrito para el caso Multioutput para modelar la característica work-stealing (figura 2.28). Utilizando las ecuaciones 2.13, 2.14 y 2.15, calculamos $T_{ser_j}$, esto es, el tiempo medio de servicio efectivo que un pipeline interno necesita para procesar, de media, los $nsubitems$ por token externo. Con esto, calculamos $T^N_{ser}$ (y $T^N_{arr}$, eq. 2.17) para la cola global $M/M/c/N/K$ que representan el pipeline externo (figura 2.27). Podemos utilizar las ecuaciones del modelo cerrado, más específicamente las ecuaciones 2.5, 2.6 y 2.7, pero reemplazando $T^{TBB}_{ser}$ y $T^{TBB}_{arr}$ por $T^N_{ser}$ y $T^N_{arr}$, respectivamente, para calcular $\lambda^N_e$, el cual es el throughput efectivo para procesar un elemento en nuestra cola global.

De nuevo, el tiempo estimado de ejecución de la implementación dependerá del throughput efectivo para procesar un elemento ($\lambda^N_e$) y el número total de elementos ($nitems$) procesados, como vemos en la ecuación 2.18.

$$Time^N = \frac{nitems}{\lambda^N_e} \tag{2.18}$$

### 2.6.4. Algunas consideraciones adicionales

Consideramos otro parámetro en el presente estudio: $\rho$, el factor de utilización. Es utilizado para modelar con que eficiencia se gestionan los servidores. Normalmente los rangos van desde 0 a 1 (ideal). Este parámetro puede ser calculado para cada caso como mostramos a continuación.

$$\text{Caso 1}: \quad \rho^H = \lambda^H_e \cdot \frac{T^{TBB}_{ser} + T^{Pth}_{ser}}{nthreads} \tag{2.19}$$

$$\text{Caso 2}: \quad \rho^M = \lambda^M_e \cdot \frac{T^M_{ser}}{nthreads} \tag{2.20}$$

$$\text{Caso 3}: \quad \rho^N = \lambda^N_e \cdot \frac{T^N_{ser}}{nthreads} \tag{2.21}$$

Es preciso resaltar una cuestión que nuestros modelos no consideran de forma inmediata: en caso de que haya filtros series en el pipeline, aparecerá un cuello de botella en algún punto. Esta limitación puede ser fácilmente modelada añadiendo una nueva restricción en nuestros modelos utilizando la siguiente ecuación,

$$\lambda_e = \frac{1}{\sum_{i\ serial} T_{ser_i}}, \quad Si \sum_{i\ serial} T_{ser_i} > \frac{1}{\lambda_e} \tag{2.22}$$

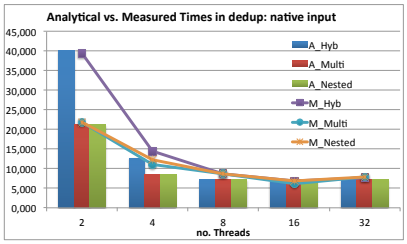
donde $T_{ser_i}$ es el tiempo necesario para procesar un elemento en el filtro serie TBB, $F_i$. Esta ecuación tiene que ser instanciada para cada caso usando $\lambda_e$ en lugar de $\lambda_e^H$, $\lambda_e^M$ o $\lambda_e^N$, según corresponda.

### 2.6.5. Validez de nuestros modelos

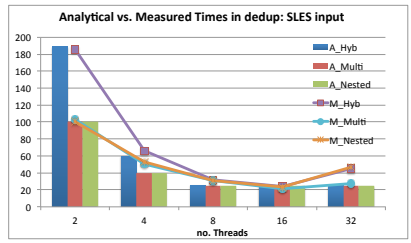
En la figura 2.29 comparamos los tiempos que resultan del modelo analítico respecto al tiempo medido en nuestra plataforma para los códigos reales. Representamos los tiempos analíticos con barras. `A_Hyb` para el caso 1 (la implementación Híbrida TBB pipeline+Pthread, eq. 2.2), `A_Multi` para el caso 2 (pipeline TBB basado en nuestro filtro Multioutput , eq. 2.16) y `A_Nested` para el caso 3 (pipeline TBB basado en pipelines anidados, la ecuación 2.18), respectivamente. `M_Hyb`, `M_Mul` y `M_Nested` representan los tiempos medidos. Las figuras muestran los resultados para tres tamaños de entradas para el problema: a) entrada native de PARSEC (672MB); b) archivo ISO del SLES11 (2.7GB); y c) fichero XML de la Wikipedia (6.74GB).

Como se muestra en la figura, los tiempos analíticos parecen predecir de manera exacta los tiempos medidos para estas tres implementaciones reales, validando así nuestro modelo analítico. En general, nuestras estimaciones analíticas tienden a dar un tiempo inferior porque como mencionamos no consideramos algunos overheads. La variación en el rango de tiempos está entre el 2 % y el 18 % hasta 16 threads, donde en todos los casos se encuentra el cuello de botella debido al filtro serie de Entrada/Salida . Sin embargo, nuestro modelo del pipeline Anidado no captura la ineficiencia de drenado en serie discutida al final de 2.5.3. Este efecto es el principal factor que explica la degradación del speedup en la implementación en 32 threads para las entradas SLES y XML, como discutimos en la sección previa.

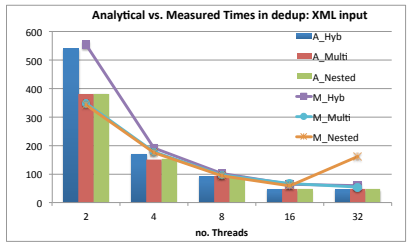
En la figura 2.29 podemos ver que para todos los datos de entrada, el modelo analíticos de la versión Multioutput y la versión Anidada producen resultados similares. Esto no es sorprendente, porque como hemos visto, ambos comparten un modelo analítico equivalente, siendo el tamaño del buffer interno ($N'$) la única diferencia entre ellos.




(a) entrada native




(b) entrada SLES




(c) entrada XML

Figura 2.29: Tiempo analítico vs. tiempo medido para todas las implementaciones de `dedup`.

En [57] demostramos que cuando el tamaño de la cola en un modelo cerrado es suficientemente grande, entonces el throughput del sitema tiende a un valor asintótico, y esto es lo que ocurre para `dedup` en las implementaciones Multioutput y Anidada, donde $N' = ntokens \cdot nsubitems$ y $N' = ntokens \cdot ntokens\_in$, respectivamente (ambos bastante grandes). Una diferencia menor es que en la implementación Anidada el tiempo necesario para realizar la escritura de un chunk se incluye como parte del tiempo de servicio de un filtro externo, en vez de considerarse como parte del tiempo de servicio medio efectivo del filtro multioutput junto con los siguientes filtros que necesitan procesar los subitems. De este modo, $T_{ser_{nb}}$ es incluido en la suma de los términos de la eq. 2.17 en vez de ser parte de $T_{ser_j}$ como ocurriera en el modelo Multioutput (eqs. 2.12 y 2.15).

Esta consideración produce rendimientos ligeramente inferiores y además, algo peores que los estimados para las implementación anidada: la degradación es siempre inferior al 1 % del tiempo estimado para la implementación multioutput. Por otro lado, los modelos analíticos también muestran claramente que ambas implementaciones mejoran a la Híbrida, especialmente cuando el número de threads es pequeño. Veamos en más detalle las razones de estas diferencias.

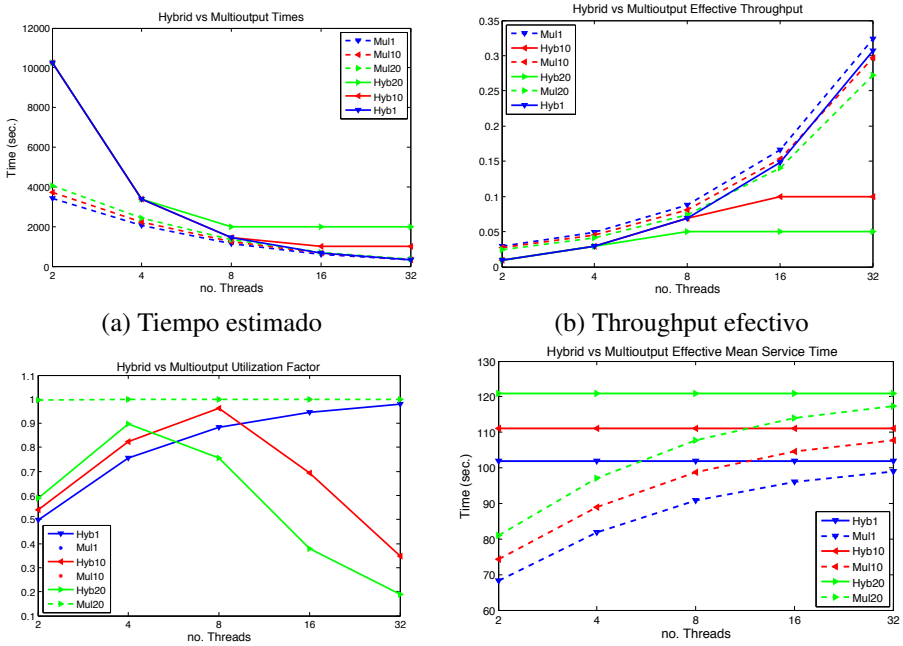
### 2.6.6. Discusión de los parámetros que afectan al rendimiento.

En la sección previa, vimos que los modelos Multioutput y Anidado se comportan de manera similar. En esta sección además, discutiremos el modelo Multioutput respecto a el modelo Híbrido. Sin embargo, recordamos que la implementación Anidada tiene un overhead adicional no considerado en el modelo analítico, ya discutido en la sección 2.5.3.

Para entender mejor por qué la implementación de pipeline basadas en el filtro multioutput es más eficiente que la Híbrida, modelamos una versión simplificada del pipeline en el cual tenemos tres filtros. $F_{in}$, $F_{work}$ y $F_{out}$. $F_{in}$ representa un filtro estándar, $F_{work}$ es un filtro que produce múltiples salidas y $F_{out}$ es el último filtro. Modelamos el comportamiento para nuestras dos implementaciones con diferentes tiempos de servicio para cada filtro, diferente número de subitems generados por el filtro $F_{work}$ y diferente número de threads. También consideramos el caso ideal para el cual la restricción 2.22 no se aplica, es decir, ninguno de los filtros es serie así que no aparecerá ningún cuello de botella debido a esto. Nuestra meta, es entender cual es el principal parámetro que afecta al rendimiento en cada implementación. De estos estudios, encontramos que ni el número de subitems ni el tiempo de servicio de $F_{in}$ tiene algún impacto en los resultados. Sin embargo, el tiempo de servicio del último filtro $F_{out}$ y el número de threads son definitivamente dos factores que contribuyen al rendimiento.

Por ejemplo, la figura 2.30(a) muestra que el tiempo analítico para las dos implementaciones cuando $T_{ser_{in}} = 1$ segundo, $T_{ser_{work}} = 100$ segundos y $T_{ser_{out}} = 1, 10, 20$ segundos, es decir, el último filtro representa aproximadamente un 1 %, 10 % y 20 % del tiempo necesario para procesar un elemento a través de un pipeline completo. Los tiempos se calculan para $nitems = 100$, $ntokens = 4 \cdot nthreads$ (el cual es la capacidad óptima en este ejemplo, siguiendo el procedimiento descrito en [57]), y $nsubitems = 20$. Las líneas solidas representan el tiempo estimado para la implementación Híbrida (Caso 1, eq. 2.2), y las líneas discontinuas representan el tiempo para la implementación Multioutput (caso 2, ecuación eq. 2.16). Los tiempos para la implementación Híbrida son siempre más altos, especialmente cuando el tiempo de servicio del filtro $F_{out}$ se incrementa. Recordemos que la implementación Híbrida de este filtro es servida por un thread dedicado, el cual serializa la ejecución de subitems a través de ese filtro. De hecho, este filtro comienza a ser el cuello de botella para el pipeline Híbrido cuando el número de threads es 16 y $T_{ser_{out}}$ representa el 10 % del tiempo necesario para procesar un elemento, mientras que el cuello de botella se manifiesta ya con 8 threads si $T_{ser_{out}}$ representa el 20 % del tiempo para procesar un elemento. Podemos ver también que el incremento de $T_{ser_{out}}$ afecta ligeramente al rendimiento de la implementación Multioutput: el tiempo analítico se incrementa un 8 % cuando $T_{ser_{out}}$ va desde 1 a 10, y se incrementan otro 9 % adicional cuando $T_{ser_{out}}$ va desde 10 a 20. Sin embargo, en

general, los tiempos para la versión Multioutput escalan cuando el número de threads se incrementa.




(a) Tiempo estimado

(b) Throughput efectivo

(c) Factor de utilización

(d) Tiempo medio de servicio

Figura 2.30: Comparación de los modelos analíticos de las implementaciones Híbrida (Hib) vs. Multioutput (Mul).

La figura 2.30(b) muestra el throughput efectivo para las dos implementaciones. De nuevo, las líneas solidas representan el throughput efectivo para la implementación Híbrida (caso 1, eq 2.1), mientras las líneas punteadas son el throughput efectivo para la implemetación Multioutput (caso 2). Podemos ver claramente el efecto del cuello de botella debido al thread dedicado en la versión Híbrida (cuando $T_{ser_{out}}$ es 10 y 20). Otro resultado interesante es que el throughput de la implementación Multioutput es menos susceptible de variar cuando $F_{out}$ cambia y el número de threads es más pequeño; cuando el número de threads se incrementa, las variaciones en el throughput efectivos son más visibles.

Para entender mejor porque el throughput efectivo es siempre más pequeño en la implementación Híbrida, calculamos el factor de utilización, $\rho^H$ y $\rho^M$, para cada caso. Esos factores pueden ser calculados por las ecuaciones 2.19 y 2.20, respectivamente.

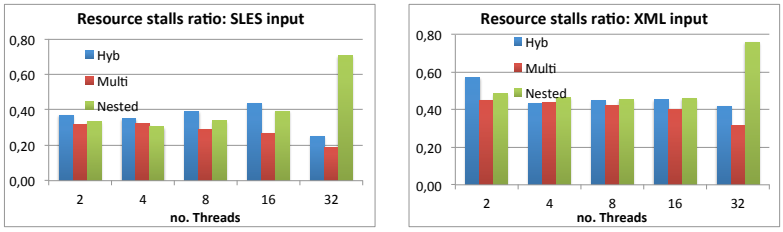
Además, en la figura 2.30(c) mostramos el factor de utilización para las dos implementaciones de nuestros experimentos. Otra vez, las líneas solidas representan los valores para la implementación Híbrida (caso 1, eq. 2.19), mientras que las líneas punteadas son para la implementación Multioutput (caso 2,eq. 2.20). Los valores para todas la variantes Multioutput están muy cercanos (representadas por la misma línea en `Mul20`). Claramente, la implementación Multioutput obtiene un factor de utilización cercano a del 1 (por tanto, cercano al 100 % de uso de los threads, es decir, un uso óptimo). Sin embargo la implementación Híbrida logra valores inferiores, especialmente cuando el cuello de botella se alcanza. Cuando $T_{ser_{out}}$ no representa un cuello de botella (la línea sólida azul, $T_{ser_{out}}$=1), podemos ver que el factor de utilización se incrementa con el número de threads y de hecho, tiende asintóticamente a los valores de la versión Multioutput. Debido a esto, con un alto número de threads, el throughput efectivo tenderá al óptimo. Esta es la razón por la que el tiempo de la implementación Híbrida se acerca al tiempo de la implementación Multioutput cuando $T_{ser_{out}}$=1.

En cualquier caso, la implementación Multioutput siempre mejora a la versión Híbrida. Basándonos en las comparaciones previas del factor de utilización, la conclusión fundamental es que, para una implementación pipeline basada en tareas, es mejor potenciar el paralelismo de grano fino (como hacemos en la versión Multioutput permitiendo procesamiento concurrente de subitems por diferentes threads o tareas) que apostar por el paralelismo de grano grueso (como hace la versión Híbrida colapsando filtros y ejecutando los subitems dentro de una tarea). El paralelismo de grano más fino permite más oportunidades para explotar completamente el planificador de work-stealing que ofrece TBB y mejorar la utilización de recursos, especialmente cuando el número de threads es pequeño, como podemos ver en la figura 2.30(d). Aquí, mostramos los tiempos medios de servicio efectivos para procesar un elemento en las dos implementaciones. De nuevo, las líneas solidas representan los valores para la implementación Híbrida (caso 1, $T_{ser}^{TBB} + T_{ser}^{Pth}$ eqs. 2.3, 2.8), mientras las líneas punteadas son para la implementación Multioutput (caso 2, $T_{ser}^{M}$, eq. 2.11). Como podemos ver, hay una reducción significativa del tiempo efectivo que la implementación Multioutput emplea para procesar un elemento, particularmente cuando el número de threads es pequeño. Esto es debido al efecto de work-stealing (balanceo dinámico) de las tareas que trabajan concurrentemente obteniendo subitems del buffer interno `delayed`. Un interesante resultado es que el efecto del work-stealing es más pronunciado en esta parte del pipeline cuando el número de threads es pequeño. Esto explica porqué en la implementación Multioutput el factor de utilización está cercano a 1 (óptimo), incluso para un número pequeño de threads, contrariamente a lo que ocurre en la implementación Híbrida y porqué la implementación Multioutput es particularmente más eficiente en estos casos.

## 2.7. Evaluaciones adicionales

### 2.7.1. Factores que afectan al overhead

En esta sección evaluamos en más detalle los principal es factores que contribuyen al overhead de las diferentes implementaciones. Nuestra metodología consiste en la instrumentalización de las diferentes implementaciones utilizando PAPI (un librería bien conocida para obtener rendimiento de contadores hardware a bajo nivel [56]) para computar algunas ratios interesantes en nuestra plataforma. Una de estas ratios es `PAPI_RES_STL/PAPI_TOT_CYC`, es decir, la relación ente los ciclos en los que hay recursos del sistema parados, respecto al total de ciclos. La figura 2.31 muestra esta ratio para los dos tamaños de entrada de `dedup` para los que se midió un mayor overhead en los experimentos previos (ver sección 2.5.3): a) El archivo ISO de SLES11 (2.7GB); y b) el archivo XML de la Wikipedia (6.74GB). `Hyb`, `Multi` y `Nested` representan la ratio de las paradas de los recursos para la implementación Híbrida, Multioutput y Anidada respectivamente.




(a) entrada SLES (b) entrada XML

Figura 2.31: Ratio de stalls en recursos para las tres implementaciones de `dedup`.

Basándonos en estas medidas podemos confirmar las predicciones hechas por nuestros modelos en la sección anterior. La utilización de recursos es menor en la implementación Híbrida, en la cual la ratio de stalls es más alta para ambos tamaños de entrada. Esto era lo esperado ya que como explicamos en las secciones anteriores, la implementación Híbrida ofrece menos oportunidades de explotar los recursos mediante el planificador work-stealing. También, en la versión Anidada encontramos mayores niveles de overhead respecto a la versión Multioutput. Esto es debido al overhead que se introduce por la creación/destrucción de los pipeline internos, así como el debido al significativo aumento del número de tareas lanzadas que tiene lugar en los pipelines internos. Más aun, la implementación Anidada presenta una elevadísima relación de stalls con 32 cores cuando se alcanza el cuello de botella de la etapa serie.

La implementación Anidada no drena eficientemente en la etapa serie de salida, ya que la implementación del pipeline interno da prioridad a completar todos los subitems del mismo token externo, en lugar de procesar primero los subitems más antiguos. Este es el principal factor que explica el incremento de la ratio de *stalls* (y también la degradación del speedup en 32 threads que vimos en la figura 2.21 para las entradas ISO y XML). Por contra, la buena utilización de recursos y mejor drenado de elementos en la etapa de salida serie explica porque los *stalls* medidos son inferiores en la implementación Multioutput para cualquier número de threads (incluso cuando hemos llegado al cuello de botella). En cualquier caso, es relativamente alta la ratio de stalls en todas las implementaciones (los recursos siempre están ociosos para más del 20 % de los ciclos) que puede ser explicado por la contención de los locks en el acceso a la tabla hash compartida utilizada en las etapas `ChunkProcess`(), `Compress`() y `SendBlock`() .

Una importante cuestión de interés es el estudio de la localidad en cada implementación. Utilizando PAPI, medimos la ratio de fallos de cache de datos para L1, L2 y LLC (último nivel de caché). Los ratios los calculamos con las siguientes formulas: `PAPI_L1_DCM/PAPI_L1_DCA` para la cache L1, `PAPI_L2_DCM/PAPI_L2_DCA` para el segundo nivel de cache y `LLC_MISES/LLC_REFERENCES` para el tercer nivel de cache, respectivamente.

Las Fig. 2.32 y 2.33 muestra la ratio de fallos para cada implementación de dedup (`Hyb`, `Multi` y `Nested`) para las entradas SLES y XML. En esta figura, podemos ver que la ratio de fallos de L1 es bastante similar en las tres implementaciones hasta 16 threads. Esta ratio se mantiene constante en todas las implementaciones (alrededor de un 2 % de ratio de fallo) con la excepción de la implementación Anidada, donde aparece un fuerte incremento que se aprecia en 16 threads y particularmente para 32 threads (cerca del 12 %). Esto ocurre tanto para la entrada ISO como para la XML.

Como dimos a entender en la sección 2.5.3, en la implementación Anidada, un pipeline interno procesa primero los subitems del mismo token incluso aunque estén pendientes subitems más antiguos de tokens previos externos que esperan ser procesados. Los elementos pendientes que están esperando en la etapa de salida se seleccionarán más tarde requiriendose más tiempo para encontrar sus entradas correspondientes en la tabla hash. Además, este problema, al cual llamamos “ineficiencia de drenado serie”, también tiene un alto impacto en los fallos de cache L1, ya que se incrementa dramáticamente el número de elementos que hay que recorrer en las listas enlazadas de la tabla hash. Recordemos que este efecto es más acentuado cuando el número de threads se incrementa, debido a que aumenta el número de elementos fuera de orden que esperan ser procesados en la etapa serie de salida.

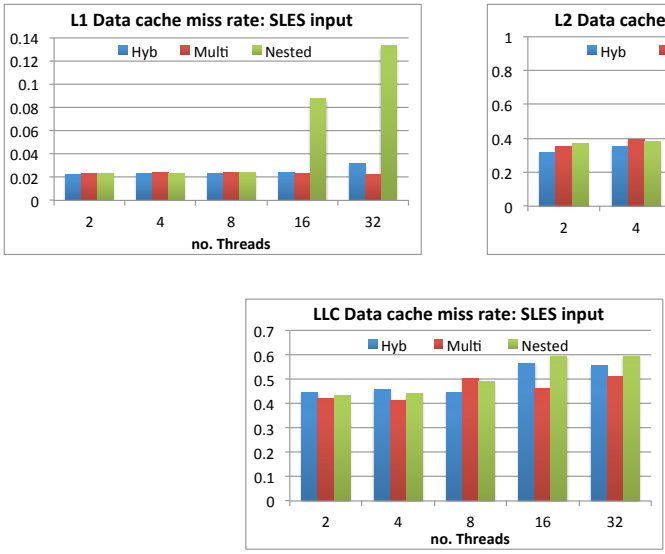



Figura 2.32: Ratio de fallos de cache L1, L2 y LLC para la entrada SLES.

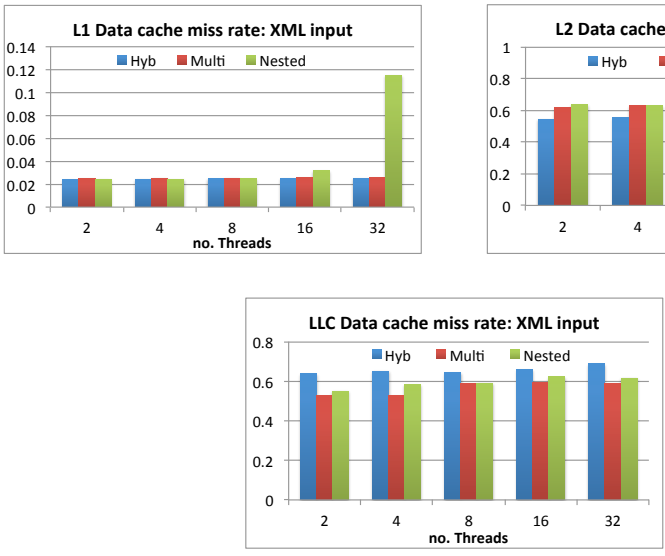


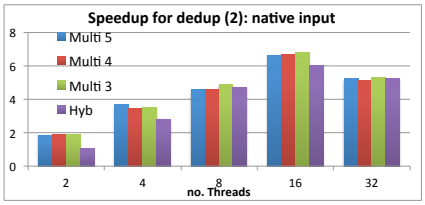

Figura 2.33: Ratio de fallos de la cache L1, L2 y LLC para la entrada XML.

Mirando la ratio de fallos de la cache de datos L2, podemos ver en las figuras que esta ratio es mayor que la ratio para L1 y de hecho, se incrementa en todas las implementaciones cuando la entrada del problema es más grande (XML). Por ejemplo, la ratio de fallos de cache L2 está alrededor del 40 % para la entrada SLES y 60 % para la entrada XML. Observamos un incremento repentino en esta ratio para la implementación Anidada cuando el número de threads se incrementa al igual que ocurría con L1, este efecto es también debido a la ineficiencia del drenado serie de los elementos que esperan en la etapa de salida.

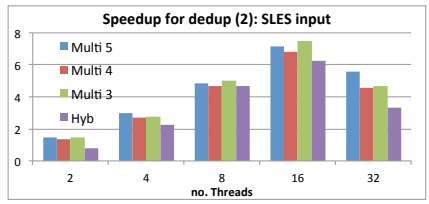
Por otro lado, la ratio de fallos de cache LLC es otra métrica interesante de rendimiento que puede indicar cuanto tráfico *off-chip* genera cada implementación. Como podemos ver en las figuras 2.32 y 2.33, el problema con una entrada de tamaño grande (XML) tiende a producir más fallos LLC y por tanto más trafico *off-chip*. Este tráfico *off-chip* tiende a incrementarse con el número de threads en todas las implementaciones. En este caso, las ratios de fallos de la cache de datos son superiores en la implementación Híbrida.

### 2.7.2. Evaluación de otras configuraciones multioutput.

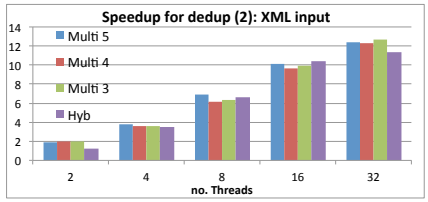
En el siguiente experimento exploramos el comportamiento de la implementación Multioutput cuando cambiamos el número de filtros y la posición de la etapa multioutput en nuestro pipeline. Nuestra metodología consiste en utilizar un filtro multioutput para la etapa `FindAllAnchors`(), pero para el resto de las etapas de dedup utilizar un número distinto de filtros. Por ejemplo, **Multi 5** es la configuración con un filtro por etapa: `DataProcess`() (el filtro de entrada serie), `FindAllAnchors`() (filtro multioutput), `ChunkProcess`(), `Compress`() y `SendBlock`() (el filtro salida serie). **Multi 4** es la configuración en la cual colapsamos las etapas `ChunkProcess`() y `Compress`() en un solo filtro paralelo y **Multi 3** es la configuración en la que colapsamos `FindAllAnchors`(), `ChunkProcess`() y `Compress`() en un solo filtro multioutput. De esta forma estudiamos el impacto cuando cambiamos el número de filtros en el pipeline. Las medidas de speedup de cada configuración en nuestra máquina se muestran en la figura 2.34. Como referencia también mostramos el speedup para la implementación Híbrida (`Hyb`). Como vemos en las figuras, las tres configuraciones tienen un comportamiento similar en los tres conjuntos de entrada estudiados. En particular, **Multi 3** tiene una escalabilidad ligeramente superior hasta 16 threads. Esto es debido a que al tener un menor número de filtros, el overhead introducido por el mecanismo de reciclado de TBB al pasar items de un filtro a otro es menor. De hecho, cuando utilizamos PAPI para calcular la ratio de stalls (`PAPI_RES_STL/PAPI_TOT_CYC`), encontramos ratios más pequeñas para **Multi 3** en los tres conjuntos de entrada, aunque la reducción de esta ratio es siempre inferior al 10 %, como vemos en al figura 2.35.




(a) entrada native

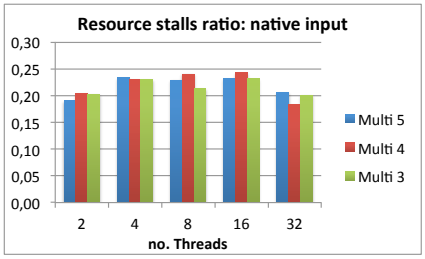



(b) entrada SLES

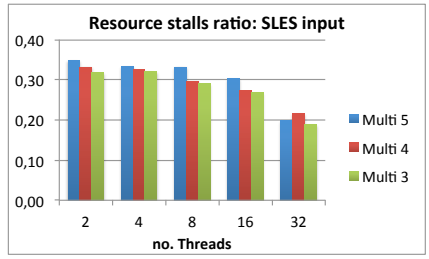



(c) entrada XML

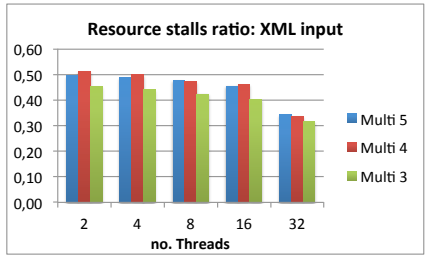
Figura 2.34: Speedup para las distintas configuraciones Multioutput de `dedup`.




(a) entrada native



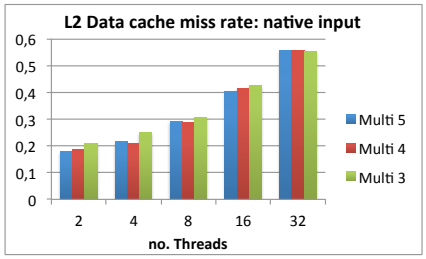

(b) entrada SLES



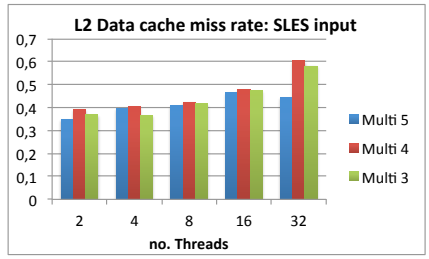

(c) entrada XML

Figura 2.35: Ratio de stalls de recursos para las diferentes configuraciones Multioutput.

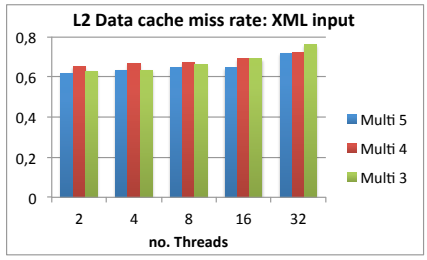
También calculamos las ratios de fallos de cache L1, L2 y LLC para las configuraciones de la implementación Multioutput, encontrando valores similares para todos los casos. De estos resultados concluímos que el número de filtros no afecta a la localidad como se muestra en la figura 2.36. La ratio de fallos de la cache L2 para las tres configuraciones y los tres conjuntos de entrada tiende a crecer conforme aumenta el número de threads en todas las configuraciones (se observa un comportamiento similar en la implementación Anidada). En otras palabras, el número de threads si afecta a la localidad. El incremento de esta ratio se debe al coste de mantener la coherencia (cache) para todas las variables globales del código dedup, ya que la cache L2 es compartida. En nuestra opinión, este es otro factor que parece afectar a la eficiencia de las implementaciones paralelas basadas en la plantilla pipeline de TBB (versiones Multioutput y Anidada).




(a) entrada native




(b) entrada SLES




(c) entrada XML

Figura 2.36: Fallos de cache L2 para diferentes configuraciones de la implementación Multioutput.

### 2.7.3. Resumen de resultados

En este estudio hemos evaluado tres implementaciones basadas en TBB de estructuras no triviales de pipeline en las que uno de los filtros puede tener más tareas saliendo de él que llegando, como ocurre en el código `dedup`. La implementación basada en el

uso de nuestro filtro multioutput ha mostrado una mejor escalabilidad que la implementación Híbrida basada en TBB + Pthreads, particularme cuando el conjunto de entrada es grande.

Hemos examinado una gran cantidad de factores que pueden explicar las diferencias de rendimiento entre ambas implementaciones:

- La ratio de stalls es más pequeña en la implementación Multioutput, porque fomenta un paralelismo de tareas de grano fino, lo que beneficia a la estrategia work-stealing para explotar mejor los recursos.
- Aun así observamos una relativamente alta ratio de stalls en la versión Multioutput debido a la contención debida al acceso a la tabla hash compartida en algunos de los filtros del código. Este overhead afecta significativamente a la eficiencia de este benchmark como podemos ver en los resultados para diferentes conjuntos de entrada.
- Por otro lado, el overhead del mecanismo de paso de tareas, que TBB utiliza para pasar un token a través de los filtros, no afecta al rendimiento de la implementación Multioutput cuando configuramos el pipeline con diferente número de filtros.
- Las ratios de fallos de cache L1 son muy bajas, mientras que para L2 se incrementan en ambos casos, aunque son similares. Sin embargo, las ratios LLC son más altas en la versión Híbrida que en la Multioutput, por tanto, la localidad de memoria se gestiona mejor en la implementación Multioutput.

Aunque desde un punto de vista analítico, las implementaciones Multioutput y la versión Anidada deberían ofrecer un rendimiento similar, encontramos que la implementación Anidada obtiene un peor rendimiento debido a:

- Un overhead adicional para la creación / destrucción de los pipelines internos, así como el overhead debido a la creación de tareas en el pipeline interno de la implementación Anidada.
- Existe un incremento en la ratio de recursos detenidos (y una consecuente degradación de rendimiento) en la implementación Anidada, cuando la Entrada/Salida alcanza el cuello de botella debido al drenado ineficiente de elementos que esperan ser procesados en la etapa de salida serie.
- La versión Anidada presenta explota peor la localidad. Observamos un incremento en los fallos de las caches L1, L2 y LLC, porque en el procesado de subitems en los pipeline internos no se da prioridad a los subitems más antiguos.

Uno de los problemas generales que encontramos en los problemas de tipo pipeline es que a partir de un cierto número de threads, y para cualquier tamaño de entrada, el cuello de botella de entrada/salida se convierte en un factor limitador de la escalabilidad. Creemos que este es un problema que debe ser abordado en trabajos futuros.

## 2.8. Trabajos relacionados

El paradigma de pipeline está recibiendo bastante atención, gracias a estar considerado dentro de las aplicaciones emergentes, en las cuales, los streaming de datos pueden ser paralelizados utilizando este modelo de manera natural. El trabajo de Raman en [66] explota una técnica llamada *Parallel-Stage Decoupled Software Pipelining (PS-DSP)*. Esta técnica, implementada en un compilador experimental, trabaja con bucles que presentan dependencias de datos, y permite identificar, con pequeñas intervenciones del programador, las etapas pipeline así como el reparto de los threads entre las etapas paralelas. Sin embargo, el algoritmo de reparto de threads se aplica en tiempo de compilación y asume una partición estática de threads para las etapas, lo que provoca que la solución no sea óptima. De hecho, uno de los principales factores que degrada la escalabilidad en sus implementaciones es el desbalanceo de carga. En los trabajos [35] y [34] realizan un estudiado del comportamiento de aplicaciones pipeline utilizando un simulador para averiguar los factores que actúan sobre el rendimiento encontrando que en el paralelismo de tareas el cuello de botella se encuentra en el camino crítico del pipeline. Para mejorar estos resultados, en cuanto a la latencia se refiere, intentan reducir el camino crítico dividiendo la etapa más costosas en pequeñas etapas en lugar de colapsar etapas como hemos propuesto nosotros. Por otro lado, estudian el mapeo eficiente de aplicaciones pipeline en sistemas paralelos. Proponen un algoritmo de distribución de tareas del pipeline entre los procesadores. A diferencia de nuestro trabajo, ellos planifican mediante un algoritmo la distribución de las tareas en los distintos procesadores (creando clusters de tareas). Para ello, tienen que examinar todos los caminos del pipeline mientras que nosotros nos abstraemos de la estructura del mismo y confiamos en el planificador work-stealing de TBB para solucionar los problemas de balanceo de carga entre las distintas etapas en sistemas memoria distribuida.

Thies propone en [79] un conjunto de anotaciones simples para permitir al programador expresar las fronteras del pipeline (las etapas) y con la ayuda de un análisis dinámico rastrea la comunicación de datos entre las etapas, lo que proporciona un diagrama de flujo de la aplicación que puede ayudar al programador a mejorar el paralelismo y el balanceo de carga. Rul en [73] mejora su trabajo, detecta automáticamente y paraleliza un pipeline en aplicaciones sin ninguna anotación. Ambas herramientas necesitan muchas horas para ser ejecutadas y consumen una gran cantidad de memoria incluso para pro-

gramas pequeños. Las aplicaciones detectan los límites de la plantilla de pipeline basada en acceso de datos. Los programas generados por estas herramientas están en forma binaria, siendo difícil examinarlos para paralelizar determinadas secciones. Sin embargo, estas herramientas son útiles para empezar a sugerir puntos donde existe paralelismo de pipeline.

Liao en [49] propone un modelo para el estudio de rendimiento de un sistema de pipeline paralelo. Sin embargo, solo se estudia el caso de pipeline paralelo abierto, y aunque considera colapsar etapas, no contempla estrategias para resolver el desbalanceo de carga, que precisamente es el principal objetivo de nuestro trabajo. El modelo del pipeline paralelo de sistemas cerrado, así como el modelo analítico para el work-stealing en el contexto del paradigma de pipeline son dos contribuciones importantes de este capítulo.

Otro enfoque para las aplicaciones de streaming es escribir códigos en un lenguaje de programación que soporte streaming como por ejemplo los lenguajes StreamC [22], StreamIt [78] o Brook [11]. Estos lenguajes son de especial aplicabilidad en procesadores de Stream [21], o en procesadores gráficos [11], aunque algunos estudios han abordado también su uso para multiprocesadores [36] o incluso para la arquitectura CELL [45]. El compilador ACTOES utiliza un algoritmo de división de un grafo estático para relacionar programas de stream en un procesador de stream. En todos los casos, siempre se implementa un planificador estático de threads por etapa del pipeline, que resulta en una degradación del speedup debido al desbalanceo de la carga. Además, las soluciones basadas en Programación Lineal Entera (ILP) han sido usadas para generar particiones de aplicaciones StreamIT en GPUs [80]. Sin embargo, formular el modelo ILP requiere ciertos conocimientos avanzados sobre la arquitectura. FlexStream [40] es un runtine que se adapta dinámicamente al sistema y relaciona un grafo de stream previamente particionado de acuerdo con el número de procesadores disponibles para un sistema multicore heterogéneo. Aleen en [2] combina información de profiling y una estimación del tiempo de ejecución para predecir comportamientos dinámicos de una aplicación de stream. Esta información de profiling se usa para ajustar dinámicamente el pipeline para conseguir balanceo de carga. Se han propuestos otros enfoques basados en una máquina de aprendizaje, que tienen como objetivo la búsqueda de una asignación eficiente del paralelismo de pipeline para procesadores multinúcleo [82]. Mattheis en [52] propone un número de variantes y extensiones de work-stealing para aplicaciones pipeline para controlar la latencia en el contexto de sistemas embebidos, donde las restricciones en tiempo real son prioritarias. Sanchez en [74] presenta una implementación de un planificador para programas irregulares de pipeline, que permite desarrollar balanceo de carga dinámica de grano fino eficientemente, al tiempo que mantiene unos niveles bajos de consumo de memoria, bajo overhead y una reserva dinámica de la cola de datos que tienen en cuenta la localidad.

Nuestras aproximación se distingue de las actuales técnicas de planificación en que nosotros enfocamos nuestros trabajo al estudio de diferentes implementaciones basadas en tareas utilizando constructores de alto nivel (el filtro multioutput para la plantilla de TBB) y cómo podemos explotar la estrategia work-stealing y el reciclado de tareas de la librería de tareas TBB eficientemente para lograr el objetivo del balanceo de carga mientras reducimos el overhead.

En un trabajo más relacionado con el nuestro, Reed et al. [70] explica cómo utilizar el pipeline de TBB para codificar aplicaciones paralelas de pipeline para el PARSEC. Ellos utilizaron una implementación de pipelines anidados para codificar `dedup`; hemos utilizado esta implementación como referencia con la que comparar nuestra versión Multioutput. También presentamos modelos analíticos para las diferentes implementaciones del pipeline paralelo no trivial para los problemas con etapa multioutput. Nuestros modelos están basados en teoría de colas y proporcionan información sobre cómo se utilizan los recursos del sistema.

## 2.9. Conclusiones

En este capítulo estudiamos la implementación basada en tareas del patrón de programación pipeline. Realizamos la implementación de dos aplicaciones del conjunto de Benchmark PARSEC e identificamos dos problemas: el desbalanceo de carga y el cuello de botella de E/S. En [57] desarrollamos modelos analíticos de comportamiento de estas aplicaciones. Propusimos dos técnicas para solventar el desbalanceo de carga: colapsar etapas y utilizar estrategias de planificación dinámica basada en work-stealing e implementamos las dos aplicaciones y las utilizamos para comprobar la validez de los modelos analíticos propuestos.

A continuación abordamos la implementación basada en tareas del problema no trivial del pipeline paralelo en el cual alguna de sus etapas pueden producir más elementos de salida que elementos de entrada. Actualmente, la plantilla de pipeline de TBB no proporciona herramientas para soportar este tipo de etapas. Específicamente, hemos diseñados y desarrollado un filtro multioutput que hemos incorporado a la plantilla de TBB para poder implementar fácilmente con este tipo de etapas. Gracias a este nuevo filtro, el usuario puede codificar de forma productiva esas estructuras de pipeline no triviales como ocurre en `dedup`. Hemos comparado el rendimiento de nuestras implementaciones de la aplicación `dedup` utilizando el filtro multioutput con otras implementaciones que utilizan solo los filtros estándar de TBB. También hemos desarrollado modelos analíticos para cada implementación con el objetivo de entender mejor la utilización de los recursos en cada caso.

Nuestra evaluación del rendimiento y los estudios analíticos nos ayudan a entender por qué la implemetación basada en el filtro multioutput mejora las otras soluciones: i) fomenta el paralelismo de grano fino de tarea para explotar mejor los recursos, ii) logra un mejor aprovechamiento de la localidad de memoria, por un lado gracias a que consigue limitar el consumo de memoria, y por otro debido a que prioriza los subitems pendientes más antiguos, y iii) reduce el overhead relativo a la creación / destrucción de objetos, así como, los overhead relacionados con el lanzamiento de tareas que otras implementaciones no pueden evitar.

# 3 Estudio del modelo basado en tareas para wavefront

Como comentamos en el capítulo introductorio, creemos que el modelo basado en tareas es más apropiado para implementar problemas de tipo wavefront, debido a que la creación y gestión de los threads es más costosa que la creación y gestión de las tareas. Además, el modelo basado en tareas planifica a alto nivel y, con la información que dispone, puede distribuir el trabajo entre los procesadores disponibles de manera eficaz. Sin embargo, como hemos visto, existen distintos paradigmas de programación basados en tareas y queremos ver cuál es el que mejor se adecua a este tipo de problemas, para posteriormente desarrollar una plantilla de alto nivel que ayude a los programadores a hacerles frente.

En este capítulo describiremos en primer lugar un problema sintético de wavefront 2D (una simplificación del problema Smith-Waterman que explicamos más adelante), ideado para poder realizar un estudio exhaustivo con distintas configuraciones del mismo. El resultado de este estudio desvela cual de las librerías o lenguajes ofrece mejor rendimiento en lo que al patrón wavefront se refiere. (sección 3.1).

A continuación, para este problema wavefront 2D, mostramos las particularidades de las implementaciones propuestas, las cuales están basadas en TBB, CnC, OpenMP y Cilk. También discutimos los resultados experimentales y los análisis comparativos de las distintas implementaciones, resultando que la versión basada en TBB es la más competitiva.

A partir de estos resultados, en la sección 3.2, explicamos cómo realizar varias optimizaciones a la implementación TBB. Esto se traduce en una mejora sustancial de esta versión de TBB de forma que aún se aumenta más la diferencia de rendimiento con respecto a las demás implementaciones.

Por último, en la sección 3.3 presentamos una plantilla basada en TBB para implementar problemas de tipo wavefront, los resultados experimentales obtenidos al usar esta alternativa para codificar varios problemas reales, así como un análisis del overhead debido a la plantilla.

## 3.1. El modelo basado en tareas aplicado a problemas de tipo wavefront

Con la intención de comparar los distintos modelos de programación basados en tareas descritos anteriormente, hemos seleccionado un problema simple de wavefront 2D. Es un problema clásico consistente en el cálculo de una función por cada celda de una matriz 2D de dimensiones $n \times n$. Este problema tiene dependencias de datos entre los elementos adyacentes a cada celda, como se muestra en la figura 3.1. Para calcular el elemento (1,1) necesitamos los datos del elemento (0,1), al norte, y del elemento (1,0), al oeste; o, dicho de otra forma, para calcular el elemento A[i,j] necesitamos tener previamente calculados el elemento A[i-1,j] y el A[i,j-1].

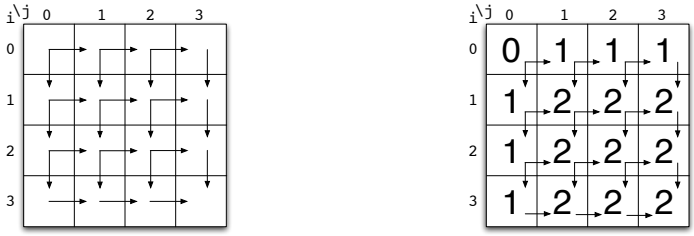
```
1 for (i=1; i<n; i++)
2   for (j=1; j<n; j++)
3     A[i,j] = foo(A[i,j], A[i-1,j], A[i,j-1], gs);
```

Figura 3.1: Código de un problema clásico de wavefront 2D.

Dado que pretendemos analizar las distintas alternativas del modelo de programación basado en tareas, tenemos que decidir cual será el trabajo que realizará la unidad básica: una tarea. Este trabajo se corresponderá con el cálculo de cada elemento o celda (i,j) de la matriz. Por tanto, nuestra intención es la de paralelizar los bucles i y j. La función `foo` de la figura 3.1 tiene un parámetro, `gs`, que sirve para controlar cuanta computación realiza esta función interna. Un valor más alto de `gs` significará una mayor carga computacional; o, lo que es lo mismo, simulará un grano de tarea más grueso. Esto nos permitirá controlar la granuralidad de la tarea y poder realizar un estudio intensivo para evaluar el rendimiento en cada caso. También podemos configurar el problema para que cada celda tenga un valor distinto de `gs`, de modo que todas las tareas no posean la misma carga computacional, y así poder evaluar el funcionamiento del work-stealing tanto con una distribución del trabajo homogénea como en un escenario de carga totalmente desbalanceado.

### 3.1.1. Implementación

En la figura 3.2(a) las flechas muestran las dependencias de datos en este problema básico de wavefront. Por ejemplo, después de ejecutar la primera tarea en la esquina superior izquierda (0,0), la cual no depende de ninguna otra, se pueden ejecutar dos tareas nuevas, la siguiente a la derecha (0,1) y la inmediatamente inferior (1,0). La información de dependencias se almacena en una matriz de contadores, donde para cada una de las celdas se guarda el número de tareas de las que depende. Sólo cuando el contador de una tarea alcance el valor nulo, ésta podrá ser creada para posteriormente ser ejecutada. Para el caso de estudio podemos ver la matriz de contadores en el estado inicial del problema en la figura 3.2(b).




(a) Flujo de dependencias de datos (b) Matriz de contadores iniciales (`counter`)

Figura 3.2: Dependencias del problema wavefront y matriz de contadores asociada.

```
1 Task_Body(); //Trabajo de la tarea
2
3 Critical Section
4 {
5   counter[i+1][j]--; //Decrementar vecino del sur
6   if (counter[i+1][j]==0)
7     Spawn();
8 }
9
10 Critical Section
11 {
12   counter[i][j+1]--; //Decrementar vecino del este
13   if (counter[i][j+1]==0)
14     Spawn();
15 }
```

Figura 3.3: Pseudocódigo de una tarea.

Resumiendo, cada tarea preparada puede comenzar a ejecutar su trabajo propio definido en el método `execute`. Una vez ejecutado todo el trabajo por la tarea, ella será la encargada de decrementar el contador de cada una de las tareas dependientes. Si alguno de estos contadores toma el valor 0, la tarea procederá a crear la nueva tarea asociada a este contador.

En el pseudocódigo (línea 1) de la figura 3.3, el `Task_Body()` corresponde con el trabajo que cada tarea debe realizar. Es decir, el cálculo de la función `foo`. Es importante mencionar que el contador de una tarea puede ser decrementado por todas las tareas de las que depende. Estas tareas pueden estar ejecutándose en paralelo y decrementar el contador a la vez. Es por esto que el contador debe ser accedido en exclusión mutua para que no existan conflictos y/o carreras. Esta razón es la que motiva que en el pseudocódigo se modifique el contador dentro de las secciones críticas (desde las líneas 4 y 11 hasta las líneas 8 y 15, respectivamente).

### 3.1.2. Implementación en TBB

La principal particularidad a la hora de implementar este problema wavefront con TBB es la utilización del tipo de datos atómico. De manera que podemos construir la matriz de contadores con elementos atómicos `atomic<int> ** counter`. Así, podremos decrementar cada contador en exclusión mutua sin necesidad de incluir una sección crítica de manera transparente al usuario. Existen una gran cantidad de métodos para las variables atómicas. Por ejemplo, la operación `--counter[i][j]` decrementa el valor del contador de la tarea (i,j) a la vez que devuelve el nuevo valor. Por tanto, escribiendo la expresión `if(--x==0) accion();` garantizamos que solo una tarea ejecutará `accion`. Si comparamos las operaciones atómicas con los *locks* o cerrojos, las operaciones atómicas son más rápidas y no hay posibilidad de *deadlock* o interbloqueos. El código escrito en TBB podemos observarlo en la figura 3.4.

Para mas detalle, entre la líneas 1 y 6 de la figura 3.4, declaramos la Clase `Operation` que hereda de la clase `TBB::Task`. Como explicamos en 1.3, es necesario redefinir el método `execute` (entre las líneas 8 y 20) que es el encargado de especificar el trabajo de la tarea (línea 9). Este método sigue el esquema del pseudocódigo de la figura 3.3:

- Calcular el nuevo valor de cada celda llamando a la función `foo(...)`.
- Decrementar y comprobar que los contadores de los vecinos son 0.
- Lanzar las tareas vecinas preparadas.

A partir de ahora, a esta versión de TBB la llamaremos TBB_v1.

```
1 Class Operation: public TBB::task{
2   int i, j;
3 public:
4   Operation(int i_, int j_) :  i(i_), j(j_) {}
5   task * execute();
6 };
7
8 TBB::task * Operation::execute(){
9   A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
10
11   if (i<n-1) //Hay vecino del sur?
12     if (--counter[i+1][j]==0)
13       spawn( * new(parent()->allocate_additional_child_of(* parent()))
             Operation(i+1, j));
14
15   if (j<n-1) //Hay vecino del este?
16     if (--counter[i][j+1]==0)
17       spawn( * new(parent()->allocate_additional_child_of(* parent()))
             Operation(i, j+1));
18
19   return NULL;
20 }
```

Figura 3.4: Detalles del código para la implementación en TBB (TBB_v1).

```
1 class MyBody{
2 public:
3 MyBody(){};
4 void operator()(block&b, tbb::parallel_do_feeder<block>&feeder) const{
5   int i = b.first;
6   int j = b.second;
7   if (i<n && j<n){
8     A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
9
10    if (i<n-1 && --counter[i+1][j]==0)
11      feeder.add(block(i+1, j));
12
13    if (j<n-1 && --counter[i][j+1]==0)
14      feeder.add(block(i, j+1));
15  }
16 }
```

Figura 3.5: Detalles del código para la implementación en TBB basada en parallel_do (TBB_v2).

Por otro lado, Intel propone una forma a alto nivel para implementar problemas

wavefront en paralelo, consistente en utilizar la plantilla `parallel_do` descrita anteriormente. Esta plantilla es, esencialmente, una lista de trabajos que se pueden ejecutar en paralelo, a la que se puede ir agregando más trabajo de manera dinámica con el método `feeder_add`, de la clase `parallel_do_feeder`. Hemos implementando el mismo problema utilizando esta herramienta, como ilustra la figura 3.5. Utilizando `feeder_add` (líneas 11 y 14) añadimos las nuevas tareas a ejecutar, una vez que están listas para ser ejecutadas. El primer parámetro del operador en la línea 4 es un par de enteros, donde se le pasa el valor de i y j a la próxima tarea. A esta implementación la llamaremos TBB_v2.

### 3.1.3. Implementación en CnC

Incrementando aun más el nivel de abstracción, podemos formular el código wavefront utilizando CnC. En la figura 3.6 mostramos la implementación del problema utilizando wavefront. Nuevamente utilizamos la plantilla atómica de TBB para la definir la matriz de contadores ( `atomic<int> ** counter`) y mantener la información de dependencias. Creamos una etiqueta `ElementTag` para prescribir los elementos del `step collection Operation`. Para codificar los pasos de la operación de una celda es necesario definir el método `Operation::Execute` (líneas 1 a 17 en la figura 3.6). Este método, precisamente, sigue el esquema de la figura 3.3, siendo la única diferencia notable la generación de etiquetas en lugar de realizar *spawns* de nuevas tareas preparadas (líneas 10 a 14).

```
1  int Operation::execute(const par & t, simple2D_context & c) const
2  {
3    int i = t.first;
4    int j = t.second;
5
6    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
7
8    if (i < n-1)
9      if (--counter[i+1][j] == 0)
10       c.ElementTag.put(par(i+1, j));
11
12   if (j < n-1)
13     if (--counter[i][j+1] == 0)
14       c.ElementTag.put(par(i, j+1));
15
16   return CnC::CNC_Success;
17 }
```

Figura 3.6: Detalles del código para la implementación en CnC.

### 3.1.4. Implementación en OpenMP

La versión 3.0 de OpenMP no posee el tipo atómico. Esto quiere decir que no se puede hacer el decremento y la comparación de manera atómica. La directiva `#pragma omp_atomic` tiene bastantes restricciones y tan solo permite hacer una sola operación de manera atómica:

`#pragma omp_atomic if(--x==0) accion();` no está permitido.

`#pragma omp_atomic priv = --X; if(priv==0) accion();` no se realiza de manera atómica. Por lo tanto, para garantizar el comportamiento deseado definiremos secciones críticas mediante la directiva `#praga_omp_critical` como podemos ver en la figura 3.7, desde las líneas 5 y 15 hasta las líneas 8 y 18, respectivamente. Otra diferencia entre OpenMP y TBB es que para crear una tarea en el primero de ellos, necesitamos la directiva `omp_task` (líneas 10 y 20 de la figura 3.7), que llamaremos de manera recursiva en un método. Creamos para ello un método denominado `Operation` (línea 1), que es el encargado de realizar el trabajo de una tarea y decrementar los contadores de las tareas vecinas (líneas 6 y 16).

```
1  void Operation(int i, int j){
2    bool ready;
3    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
4    if (j<n-1) {
5  #pragma omp critical{
6      --counter[i][j+1];
7      ready = counter[i][j+1]==0;
8  }
9      if (ready){
10 #pragma omp task
11       Operation(i, j+1);
12     }
13   }
14   if (i<n-1){
15 #pragma omp critical{
16     --counter[i+1][j];
17     ready = counter[i+1][j]==0;
18 }
19     if (ready){
20 #pragma omp task
21       Operation(i+1, j);
22     }
23  }
24 }
```

Figura 3.7: Detalles del código para la implementación en OpenMP basada en secciones críticas (OpenMP_v1).

Las secciones críticas que carecen de nombre son consideradas como si del mismo nombre se tratase. Por tanto, cualquier hilo que intente acceder a una sección crítica de estas características se bloqueará hasta que el thread que está dentro salga de ella. En general, un thread estará esperando antes de una sección crítica hasta que no exista ningún thread ejecutando una sección crítica del mismo nombre. A esta implementación le llamaremos OpenMP_v1 (figura 3.7). Lógicamente esta implementación basada en secciones críticas necesita un grano de tarea más grueso, ya que las colisiones serán de todos los elementos con todos, dando igual el contador que se esté decrementando. Esto podría evitarse declarando una matriz de cerrojos. Si declaramos una matriz de $n \times n$ con elementos de tipo `omp_lock_t` podemos tener un cerrojo para cada uno de los contadores, por lo que sólo tendríamos colisiones cuando dos tareas intentasen acceder al mismo contador, como en el caso de las variables atómicas. Por ejemplo, en la figura 3.8, en vez de la directiva `omp_critical` para acceder en exclusión mutua al decremento de los contadores, escribiríamos `omp_set_lock(locks[i][j+1])` (líneas 5 a 14) y pondríamos `omp_unset_lock(&locks[i][j+1])` en las líneas 8 y 17 para liberar el cerrojo.

```
1  void Operation(int i,int j){
2    bool ready;
3    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
4    if (j<n-1){
5      omp_set_lock(&locks[i][j+1]);
6        --l_counter[i][j+1];
7        ready = l_counter[i][j+1]==0;
8      omp_unset_lock(&locks[i][j+1]);
9      if (ready){
10 #pragma omp task
11       Operation(i, j+1);
12     }
13   if (i<n-1){
14     omp_set_lock(&locks[i+1][j]);
15       --l_counter[i+1][j];
16       ready = l_counter[i+1][j]==0;
17     omp_unset_lock(&locks[i+1][j]);
18     if (ready){
19 #pragma omp task
20       Operation(i+1, j);
21     }
22  }
23 }
```

Figura 3.8: Detalles del código para la implementación en OpenMP basada en locks (OpenMP_v2).

Aunque esta forma de programar emula al modelo de programación de Pthreads y se aleja de nuestra meta de ofrecer un entorno de alto nivel de abstracción para programar problemas de tipo wavefront. A esta implementación la denominamos OpenMP_v2.

### 3.1.5. Implementación en Intel Cilk plus

```
1  void Operation(int i,int j){
2    A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs);
3
4    if (j<n-1){
5      pthread_mutex_lock (&mylock[i][j+1]);
6      counters[i][j+1]--;
7      firstReady = (counters[i][j+1]==0);
8      pthread_mutex_unlock(&mylock[i][j+1]);
9
10     if (firstReady>0)
11       cilk_spawn Operation(i,j+1);
12   }
13
14   if (i<n-1){
15     pthread_mutex_lock(&mylock[i+1][j]);
16     counters[i+1][j]--;
17     secondReady = (counters[i+1][j]==0);
18     pthread_mutex_unlock(&mylock[i+1][j]);
19
20     if (secondReady>0)
21      cilk_spawn Operation(i+1,j);
22    }
23
24
25   if (firstReady || secondReady)
26     cilk_sync;
27 }
```

Figura 3.9: Detalles de implementación del código en Cilk.

Para completar nuestro estudio hemos realizado una implementación en Cilk. Concretamente utilizando el framework Intel Cilk Plus que provee Intel. El pseudo código mostrado en la figura 3.9 es muy parecido al utilizado en `TBB_v1` y `OpenMP_v2`. Sustituimos los *spawns* de TBB por `cilk_spawn` (líneas 11 y 21) y usamos el constructor `cilk_sync` (línea 26) para sincronizar todas las tareas, ya que el constructor `cilk_spawn` no espera a que las tareas creadas terminen. Otra principal diferencia viene dada porque Cilk no posee tipo atómico, por lo que utilizamos cerrojos de Pthreads (líneas 5, 8, 15 y 18) para implementar la exclusión mutua. Por lo demás, la estructura

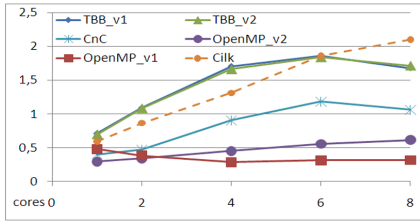
del código es siempre la del esquema principal que hemos mostrado anteriormente.

### 3.1.6. Resultados experimentales del modelo basado en tareas

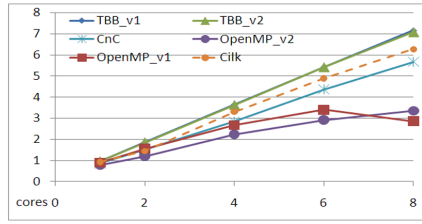
En este apartado presentamos un conjunto de experimentos para evaluar el modelo de programación basado en tareas en la paralelización de problemas de tipo wavefront. En estos experimentos hemos utilizado un sistema multicore con 8 procesadores, donde cada core es un Intel Xeon® CPU X5355 con 2,66Ghz y Suse Linux 10.1 como sistema operativo de la plataforma. Los códigos han sido compilados siempre con icc 11.1 y un nivel –O3 de optimización. Para recolectar los resultados, hemos ejecutado cada experimento 10 veces, para posteriormente, calcular la media de los tiempos recogidos. El speedup ha sido calculado respecto al tiempo de la ejecución del código secuencial. Cada experimento lo ejecutamos en 1, 2, 4, 6 y 8 cores para ver como escala cada uno de los códigos al incrementar el número de cores. Hemos dividido los experimentos en dos tipos, grano fijo (todas las tareas tienen la misma carga computacional) y grano variable (cada tarea realiza distinto número de operaciones). En el primero de ellos comparamos las distintas implementaciones para ver cuál de ellas obtiene mejor rendimiento, mientras que en el segundo analizamos los efectos del work-stealing.

#### 3.1.6.1. Grano fijo

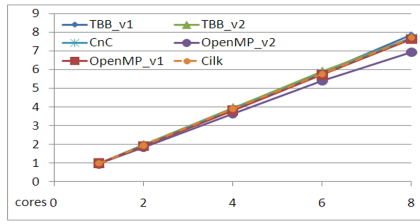
En primer lugar presentamos un conjunto de experimentos consistente en la implementación del problema básico como caso de estudio. Para este experimento hemos fijado el grano de tarea de manera constante en todas las celdas. Esto significa que la carga computacional es la misma para cada tarea y que el trabajo está distribuido de manera igual para todos los threads, excepto al principio y al final de la ejecución. El objetivo principal es comparar el rendimiento de las implementaciónes: TBB_v1, TBB_v2, OpenMP_v1, OpenMP_v2, CnC y Cilk. Concretamente TBB_v1 representa la versión de TBB que controla la creación de las tareas a través del método `spawn`, como muestra la figura 3.4; mientras que TBB_v2 representa el resultado de la versión de TBB que controla esta creación mediante el uso de la plantilla `parallel_do` y el método `feeder.add()`, como discutimos en la sección 3.1.2. El resultado del código CnC descrito en la figura 3.6 es etiquetado como CnC. Por otro lado, OpenMP_v1 representa el resultado obtenido por OpenMP y la versión basada en el uso de la directiva critical (ver figura 3.7). Mientras que OpenMP_v2 representa el resultado para la versión de OpenMP basada en el uso de cerrojos `omp_lock` (ver figura 3.8). Y finalmente, Cilk es la versión implementada con Cilk plus.




(a) Grano fino de tarea




(b) Grano medio de tarea




(c) Grano grueso de tarea

Figura 3.10: Speedup para grano de tarea constante y distinto número de cores.

El objetivo de este experimento es conocer el comportamiento de estas implementaciones con diferente tamaño de grano. El grano de la tarea lo controlamos con el parámetro `gs` de la función `foo`. Hemos propuesto tres granos diferentes que consideramos que serán grano fino, grano medio y grano grueso. Respectivamente se corresponden con a) `gs` = 100 (son aproximadamente 200 operaciones en punto flotante), b) `gs` = 1000 ( aproximadamente 2000 operaciones en punto flotante) y c) `gs` = 10000 (aproximadamente 20000 operaciones en punto flotante). En cualquier caso, el valor de `gs` será constante para todas las tareas. Por esta razón, salvo al comienzo y al final de la computación del problema, el trabajo será constante para todos los threads. Realizamos experimentos con matrices de distintos tamaños, aunque en la figura 3.10, solo mostramos los resultados con matrices de tamaño $1000 \times 1000$, porque el speedup obtenido para cada tamaño de matriz significativamente grande es siempre similar. Deberíamos remarcar que la implementación de Cilk aborta aleatoriamente. De hecho, es totalmente imposible ejecutar esta versión con matrices superiores a $1500 \times 1500$. Esto es debido a la limitación en el número de tareas anidadas que permite la implementación de Intel Cilk Plus. Esperamos que en futuras versiones de la extensión, Cilk admita más paralelismo anidado y se solucione este problema. Para el resto de versiones del problema wavefront hemos realizado los mismos experimentos con otros tamaños, obteniendo resultados similares.

En la figura 3.10 mostramos el speedup para los códigos de TBB, OpenMP, CnC y Cilk en los tres casos a, b y c. La primera conclusión que obtenemos de este experimento es que la capacidad de escalar de cada código depende del grano de tarea. Se puede observar claramente como el speedup es mejor en grano grueso y aceptable en grano medio, mientras que en grano fino el speedup es decepcionante para todos los códigos. Para grano medio y fino, TBB supera al resto de las implementaciones, excepto a Cilk con 8 cores que obtiene un mayor rendimiento. En general, Cilk tiende a escalar mejor. El problema es que no todas las ejecuciones en Cilk se ejecutan correctamente. Por otro lado, en grano grueso el comportamiento de todos los códigos es similar. Las versiones de TBB_v1, basada en `spawns` explícitos y TBB_v2, basada en la plantilla `parallel_do_feeder` alcanzan resultados similares; aunque en grano fino, el caso (a) TBB_v1 exhibe un mejor rendimiento de 1 a 6 cores. Otra conclusión importante es que el rendimiento en OpenMP se degrada más rápidamente. La versión de OpenMP_v1 basada en secciones críticas es una de las que peor rendimiento tiene. Por el contrario, OpenMP_v2, implementación basada en locks, aunque exhibe un mal resultado en el caso (a) (grano fino) para un pequeño número de cores, escala cuando el número de cores crece. Esto es debido a la penalización que tiene que pagar por la inicialización de los cerrojos. Por otro lado, CnC se mantiene entre las versiones de TBB y OpenMP.

Una vez analizado el rendimiento de las distintas implementaciones, queremos emplear la actividad *Call Graph* de Vtune. Esta herramienta trabaja analizando los puntos de entrada y salida de todos los programas y las funciones o métodos de las distintas librerías. Puede detectar módulos del programa cuando son cargados en tiempo de ejecución. El *Call Graph* proporciona una información importante sobre las funciones analizadas: el tiempo consumido, el tiempo de espera y características de llamada (quién llama y a quién llama cada función, así como el número de llamadas). Utilizamos esta actividad para recolectar información sobre las funciones que más tiempo consumen en los distintos casos: a, b y c (a son las ejecuciones con grano fino, b son ejecuciones con grano medio y c con grano grueso).

Estas medidas las realizamos para TBB_v1, TBB_v2, OpenMP_v1, OpenMP_v2 y CnC. De este modo podemos estudiar el overhead producido por las funciones de la librería TBB, de las directivas de OpenMP y de las funciones de la librería de CnC. Mostramos en la figura 3.11, 3.12 y 3.13 la gráfica con el tiempo consumido por las funciones más importantes de las librerías de TBB, OpenMP y CnC respectivamente para cada una de las versiones. En el eje de ordenadas "y" representamos la ratio entre el tiempo consumido por cada función y el tiempo total de ejecución. Para las implementaciónes TBB_v1 y TBB_v2, los métodos que más tiempo consumen son `spawn`, `allocate` y `get_task` (`spawn` es el método invocado cada vez que una tarea nueva es lanzada, `allocate` selecciona la mejor asignación de memoria disponible y `get_task` es un método del planificador de TBB llamado después de la ejecución de una tarea previa).

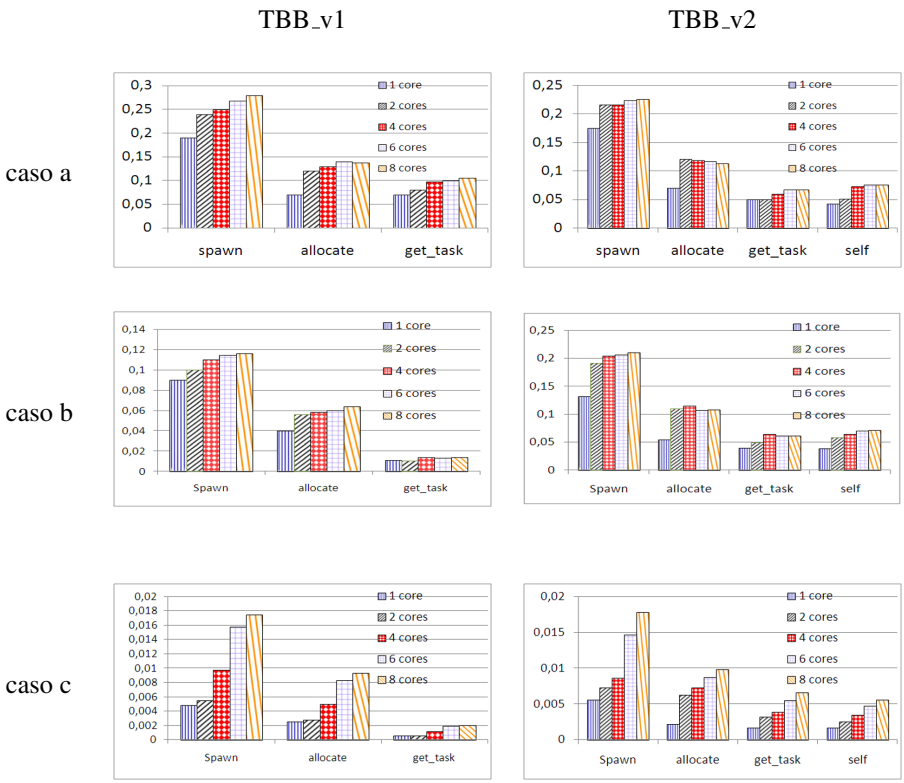



Figura 3.11: Resultados del profiling para los métodos de librerías que más tiempo consumen, para los casos de las versiones de TBB en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

Los overheads introducidos por TBB_v1 y TBB_v2 son muy similares; sin embargo, en TBB_v2 aparece un método interno adicional, `self()` (método que es invocada por `parallel_do_feeder()`), cuya contribución es grande. Cuando el grano de tarea es muy fino (caso a) o medio (caso b), el overhead introducido por `spawn` es alto; especialmente en el caso fino (caso a), ya que está cercano al 28 % del tiempo total de ejecución en TBB_v1. Sin embargo, en grano grueso (caso c) el overhead es muy pequeño. Lo cual concuerda con los resultados de la figura 3.10, donde vemos que el speedup en grano grueso es bastante bueno. De cualquier modo, el overhead aumenta siempre con el número de cores.

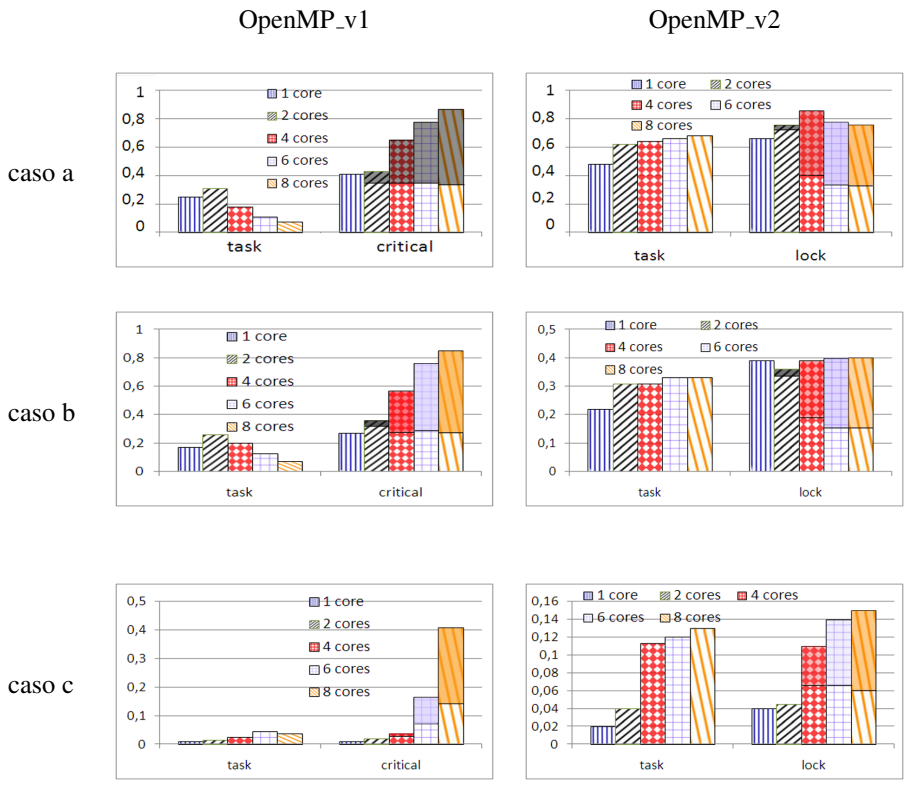
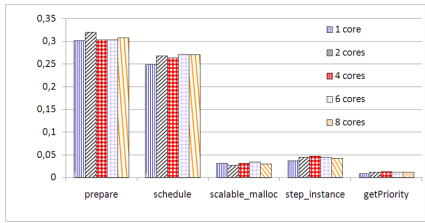


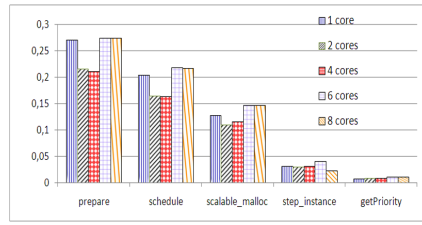

Figura 3.12: Resultados del profiling para las funciones de librerías que más tiempo consumen, para los casos de las versiones de OpenMP en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

Análogamente, se aprecia las versiones de OpenMP y observamos que las funciones específicas que consumen más tiempo para `OpenMP_v1` son el conjunto denominado *task* (las llamadas provocadas por las directivas `omp_task` y `omp_task_alloc`, asociadas a la gestión de una tarea), y el conjunto denominado *critical* (funciones asociadas a las directivas `omp_critical` y `omp_end_critical` utilizadas para acceder en exclusión mutua). En estos resultados se observa que el overhead de la creación de tareas es importante en grano fino (caso a) y está cercano al 30 % del tiempo total de ejecución. Sin embargo, el overhead introducido por la creación de tareas no se incrementa con el número de cores como ocurre en TBB. Por otro lado, el overhead introducido por la di-

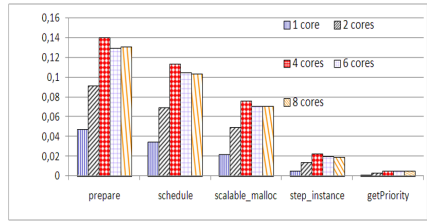
rectiva *critical* es la principal fuente de ineficiencia. Hemos sombreado la parte alta de la barra de la directiva *critical* con la contribución hecha por el tiempo de espera producido por la llamada `omp_lock_acquire`, función llamada a su vez por `omp_critical`. Precisamente, ese tiempo de espera aumenta significativamente con el número de cores en todos los casos, especialmente en grano fino donde puede llegar a alcanzar cerca del 60 % del tiempo total de ejecución. Eso explica (añadido al tiempo de creación de tareas) los pobres resultados de escalabilidad que vimos en grano fino en los códigos de OpenMP cuando incrementabamos el número de cores. El principal problema es que el tiempo de espera creado por los cerrojos puede empezar a serializar la ejecución y se convierte en un auténtico cuello de botella, incluso en grano grueso cuando el número de cores se incrementa (ver la tendencia en la figura 3.10 para el caso c).




caso a




caso b




caso c

Figura 3.13: Resultados del profiling para los métodos de librerías que más tiempo consumen, para los casos de CnC en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

En OpenMP_v2 hay también cuatro funciones implicadas divididas en dos grupos: *task* (relativas a `omp_task` y `omp_task_alloc`) y *lock* (relativas a `omp_set_lock` y `omp_unset_lock`, utilizadas para los accesos en exclusión mutua). Nuevamente, el overhead de la creación de tareas es significativamente grande en grano fino de tarea. También, hemos sombreado la parte superior de las barras con el tiempo de espera producido por las directivas `omp_set_lock` y `omp_unset_lock`. Este tiempo se in-

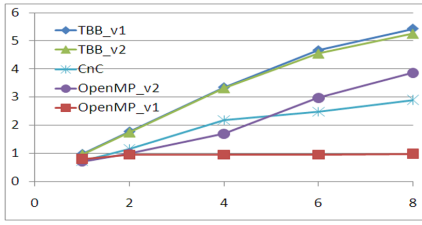
crementa con el número de cores. Sin embargo, el overhead introducido por la gestión de los cerrojos es bastante más pequeño comparado con el overhead introducido por las secciones críticas. No obstante, sigue aumentando con el número de cores.

En CnC hay algunos métodos que consumen más tiempo que el resto: `prepare`, `schedule`, `step_instane`, `scallable_malloc` y `get_priority`, que son llamados por `Put`. El método `Put` es llamado por el usuario al añadir una nueva etiqueta. El 50 % del overhead introducido por el planificador es overhead propio de TBB (llamadas a `spawns`, `allocate`, etc.). Además, el resto del overhead es resultado de la necesidad de hacer ahora más llamadas a funciones auxiliares. También es importante señalar que el overhead introducido por estos métodos tiende a disminuir conforme aumentamos el tamaño del grano de tarea, como se observa para el grano medio y grueso en CnC.

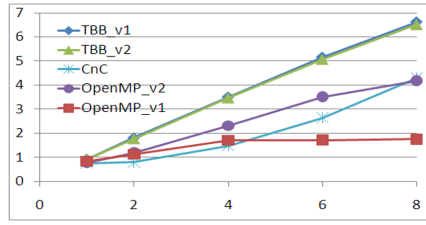
#### 3.1.6.2. Grano variable

A continuación presentamos un segundo conjunto de experimentos ideado para estudiar los efectos de una carga de trabajo desbalanceada en las seis implementaciones: TBB_v1, TBB_v2, OpenMP_v1, OpenMP_v2, Cilk y CnC. El desbalanceo de carga se consigue dándole un valor distinto al parámetro `gs` para cada celda; de esta manera cada celda realizará distinto número de FLOP y por tanto cada tarea tendrá distinta carga computacional. Nuevamente, en estos experimentos variaremos la granuralidad media de la tarea. En los experimentos anteriores demostramos que el tamaño de la matriz no es importante en este tipo de problemas. Así que seleccionamos una matriz de tamaño $1000 \times 1000$ y tres granuralidades distintas de tarea: caso 1) `gs` = [100..400] (aproximadamente entre 200 y 800 operaciones en punto flotante, siendo para nosotros el grano fino de tarea); caso 2) `gs` = [100..1000] (aproximadamente entre 200 y 2000 operaciones en punto flotante, grano medio); y caso 3) `gs` = [1000..10000] (aproximadamente entre 2000 y 20000 operaciones en punto flotante, definiéndolo como grano grueso). El speedup para estos casos está mostrado en la figura 3.14. Como ocurriera en los experimentos anteriores algo provoca que la versión de Cilk no acabe correctamente en todas las ocasiones. En los resultados observamos que la versión de TBB siempre supera a las versiones de OpenMP. OpenMP no se amolda bien con el grano variable de tarea. Especialmente en problemas con grano fino o muy fino. Sin embargo, el comportamiento de TBB parece mejor. Para grano grueso de tarea, la degradación del rendimiento para OpenMP en desbalanceo de carga no es tan evidente. La versión de CnC obtiene siempre un speedup entre TBB y OpenMP. Nuevamente el comportamiento de OpenMP_v2 es mejor que la versión OpenMP_v1. Si comparamos el grano fijo de tarea con estos experimentos observamos que las ejecuciones con grano variable de tarea son mejores que para grano fino (caso a y caso 1). La razón que justifica esto es que en grano variable

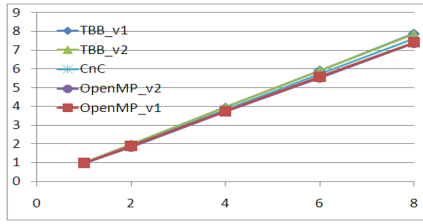
(caso 1) el valor de `gs` varia entre 100 y 400. Esto significa que la carga de trabajo en media es superior al caso a (`gs` = 100).




(a) Caso 1: Speedup




(b) Caso 2: Speedup




(c) Grano grueso de tarea

Figura 3.14: Resultados para las versiones en TBB, OpenMP y CnC para problemas con desbalanceo de carga.

Como hicimos en los experimentos de grano fijo, utilizamos Vtune para analizar las fuentes de overhead en los tres casos de grano variable. Los resultados para la mayoría de las funciones están mostradas en la figura 3.15, 3.16 y 3.17. El eje de coordenadas "y" representa la ratio entre el tiempo de cada método y el tiempo de ejecución total del problema. TBB_v1 y TBB_v2 tienen tres métodos: `spawn`, `allocate` y `get_task`. El overhead introducido por esas funciones se incrementa con el número de cores. Sin embargo, en grano fino el overhead introducido por `spawn` está sobre el 25 % en TBB_v1. En TBB_v2 el overhead introducido por `spawn` es menor. Aunque hay un método adicional: `self()`. En el caso 3, grano grueso de tarea, el overhead es muy pequeño y todas las implementaciones obtienen buenos resultados, como se aprecia en la figura 3.14.

Las funciones que consumen más tiempo para OpenMP_v1 son organizadas nuevamente en dos grupos. Aquellas relativas a las tareas, conjunto que agrupa la creación y terminación de tareas (`omp_task_allock y omp_task`) y las responsables de controlar las secciones criticas (`omp_critical y omp_end_critical`). El overhead de las tareas, como ocurre en grano fijo y fino, está cercano al 30 % del tiempo total de

ejecución. Sin embargo, en grano grueso el overhead no es significativo. Otra vez hemos sombreado la parte alta de las barras de critical con la contribución del tiempo de espera de las funciones internas `omp_lock_acquire` que son llamadas por `omp_critical`. De esta forma es apreciable como el tiempo de espera aumenta conforme aumentamos el número de cores. Con 8 cores el tiempo de espera alcanza el 60 % del tiempo total de ejecución.

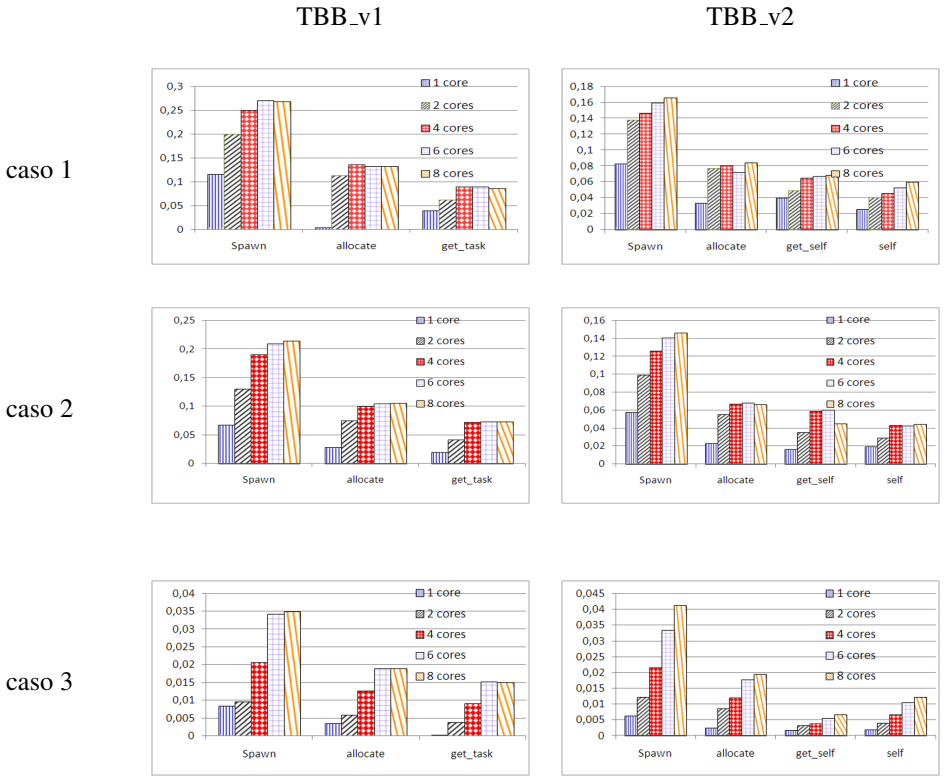



Figura 3.15: Resultados del profiling para los métodos de librerías que más tiempo consumen, para los casos de las versiones de TBB en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

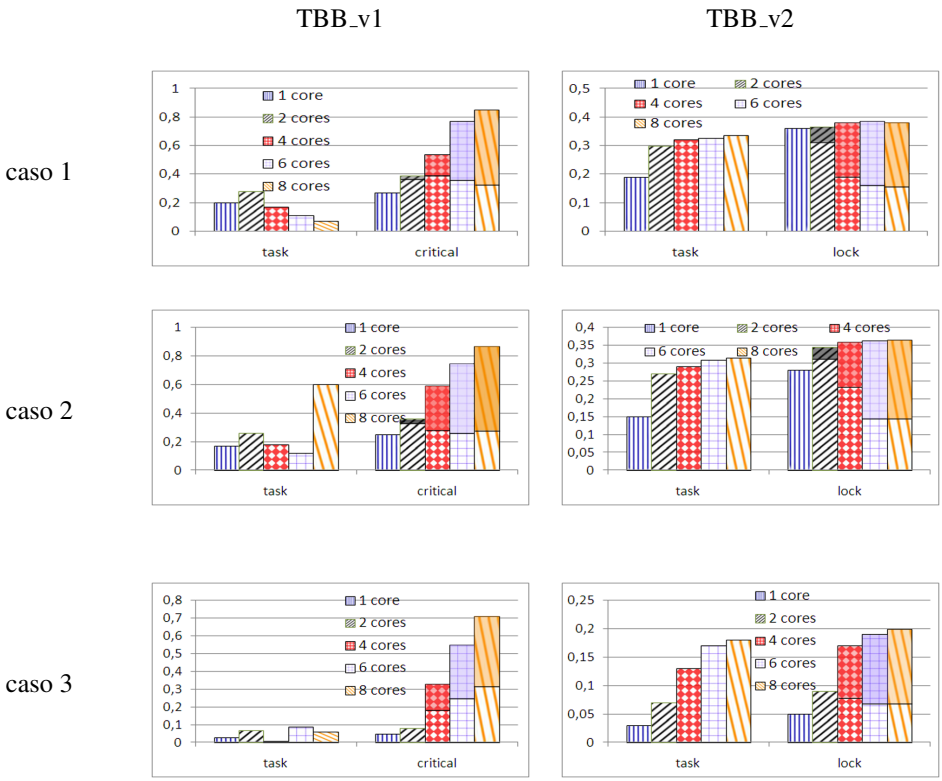
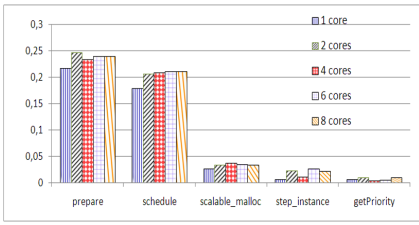



Figura 3.16: Resultados del profiling para las funciones de librerías que más tiempo consumen, para los casos de las versiones de TBB en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

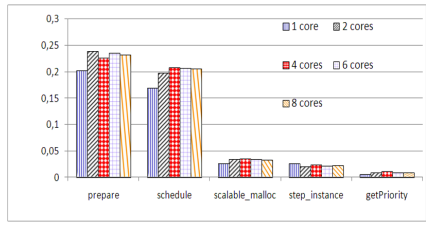
Para OpenMP_v2 tenemos dos conjuntos. La diferencia está en que hemos utilizado *locks* (`omp_set_lock y omp_unset_lock`) en lugar de *critical*. No obstante, hemos vuelto a sombrear la parte alta de la barra de los *locks* con el tiempo espera de las funciones *locks*, el cual se incrementa conforme aumentamos el número de cores. Sin embargo, la principal diferencia con OpenMP_v1 es que el overhead introducido por los cerrojos es menor.

Finalmente en CnC analizamos los métodos que consumen más tiempo en el método `Put`. Este método es el que más tiempo de espera tiene. Las funciones son nuevamente: `prepare`, `schedule`, `step_instane`, `scallable_malloc` y `get_priority`.

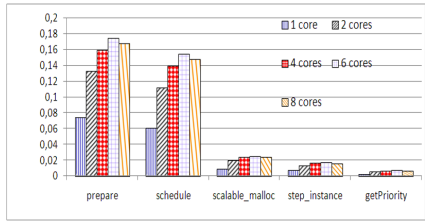
El 50 % del overhead introducido por el planificador está caracterizado por el overhead que introduce TBB (`spawns`, `allocate`, etc.).




caso 1




caso 2




caso 3

Figura 3.17: Resultados del profiling para los métodos de librerías que más tiempo consumen, para los casos de CnC en 1, 2, 4, 6 y 8 cores. En el eje de ordenadas se presenta la ratio entre el tiempo de ejecución propio de cada función y el total de ejecución.

## 3.2. Optimizaciones

A la vista de los resultados anteriores, en esta sección proponemos distintas alternativas que permitan reducir los overheads de mayor impacto. Dado que el caso de grano fino es el que presenta peor escalabilidad, dedicaremos mayor atención a la situación. Como adelanto, las tres optimizaciones propuestas se basan en a) instrucciones atómicas, b) reciclado de tareas y mejor aprovechamiento de la cache y c) tiling para explotar aun mejor la cache.

### 3.2.1. Instrucciones atómicas

Una de las complicaciones que derivan en la ineficiencia de los códigos es la ausencia de captura atómica en OpenMP 3.0. Sin embargo, esto podría ser solucionado utilizando

la instrucción de bajo nivel `compare-and-swap` (CAS). Mostramos en la Figura 3.18 una nueva versión de OpenMP basada en esta optimización, donde CompareAtomic() (líneas 3 y 6 de la figura) es ahora responsable del desarrollo de los accesos atómicos a la matriz de contadores (líneas 10 y 16).

```
1 void Operation(int i, int j)
2 {
3   int CompareAtomic(int ii, int j){
4     int x= __sync_sub_and_fetch((volatile int *) & counter[i][j], 1);
5     return x==0;
6   }
7   int gs;
8   A[i][j] = foo(A[i][j], A[i-1][j], A[i][j-1], gs));
9   if(j<n-1){
10    if(CompareAtomic(i, j+1)){
11 #pragma omp task
12      Operation(i, j+1);
13    }
14  }
15  if(i<n-1){
16    if(CompareAtomic(i, j+1)){
17 #pragma omp task
18      Operation(i+1, j);
19  }
20  }
21 }
```

Figura 3.18: Detalles del código para la implementación en OpenMP con la instrucción CAS.

Evaluamos esta nueva versión de OpenMP (llamada OpenMP_v1.2) siguiendo la misma metodología explicada en las secciones anteriores. Los resultados para el grano fino constante, caso a) (200 FLOP), se muestran en la figura 3.19, donde también reflejamos los resultados de OpenMP_v1 y OpenMP_v2 (las versiones de OpenMP basadas en sección crítica y en el uso de cerrojos respectivamente); así como también la versión de TBB_v1 (la versión de TBB con accesos atómicos utilizando la plantilla apropiada para ese cometido) para poder comparar directamente. De este modo podemos ver como la sincronización basada en la instrucción CAS reduce claramente los problemas de contención en OpenMP, haciendo que la versión optimizada de OpenMP sea la mejor a partir de 6 a 8 cores. Para el grano medio de tarea TBB_v1 sigue siendo significantemente mejor su rendimiento, mientras que para grano grueso los resultados son prácticamente idénticos.

Esta vez utilizamos Vtune para analizar los eventos y recoger el impacto de las operaciones relativas al uso de los cerrojos, con el propósito de medir los porcentajes de

ciclos gastados por las esperas derivadas. Encontramos que en el caso de grano fino la contención creada por las operaciones atómicas está cercana al 30 % de los ciclos de CPU para 6 y 8 cores en TBB_v1, mientras que OpenMP_v1.2 obtiene una ratio alrededor del 20 % que explica por qué OpenMP optimizado mejora significativamente el resultado de TBB_v1. Sin embargo, en el grano medio la ratio es siempre inferior al 5 % para la versión de TBB_v1, encontrando que OpenMP_v1.2 es superior al 6 % explicando las pequeñas diferencias que aparecen en el speedup. En el caso grueso las ratios se vuelven indiferentes, ya que no se aprecia prácticamente un porcentaje tangible. De todos modos, estos resultados demuestran que la captura atómica es una importante característica que debería estar disponible a alto nivel. De hecho, esta captura atómica estará implementada en OpenMP 3.1.

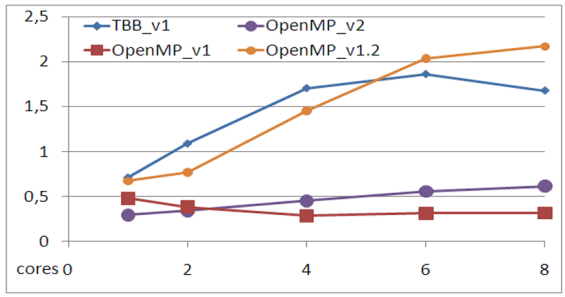



Figura 3.19: Speedup para la versión optimizada en OpenMP basada en instrucciones CAS comparado con la versión TBB_v1.

### 3.2.2. Reciclado y guía para el planificador

Los resultados de la sección anterior muestran que parece difícil reducir el overhead introducido por las operaciones atómicas; por tanto, los desarrolladores de las librerías de OpenMP y TBB deberían intentar reducir otras fuentes de overhead: el coste de la creación y gestión de las tareas, uno de los principales problemas para conseguir un mejor rendimiento cuando el grano de tarea no es suficiente.

Un camino para el programador puede ser reducir el coste de la creación de tareas y el manejo de las mismas utilizando explícitamente el mecanismo de reciclado de tareas. En nuestro patrón wavefront, cada tarea tiene la oportunidad de lanzar dos nuevas tareas (los vecinos este y sur). Podemos evitar el `spawn` de uno de ellos devolviendo un puntero a la siguiente tarea. De este modo, lanzaríamos una nueva tarea y nos reciclaríamos en la primera disponible. Esta estrategia logra dos objetivos: reduce el número de llamadas al método `spawn()` y ahorra tiempo al no tener que obtener una nueva tarea de la cola de

tareas de cada thread (reduciendo el número de llamadas a `get_task()`). En OpenMP o Cilk, este mecanismo de reciclado podría emularse llamando recursivamente al cuerpo de cada tarea en lugar de lanzar una nueva, evitando así el lanzamiento de la misma. Sin embargo, esta estrategia de emulación causa problemas de anidamiento en la pila, ya que las llamadas recursivas son muy numerosas, un problema que el mecanismo de reciclado de TBB evita. Además, cuando reciclamos podemos ayudar al planificador informándole sobre qué tarea priorizar a la hora de reciclar, garantizando que exista mayor aprovechamiento de la caché al recorrer la estructura de datos, mejorando así la localidad espacial.

En la figura 3.20 podemos ver los códigos con la optimización implementada en TBB.

```
1  if (j<n-1){
2    if (--counters[i][j+1]==0){
3       recycle_into_east = true;
4    }
5  }
6  if (i<n-1){
7    if (--counters[i+1][j]==0){
8       if (!recycle_into_east){
9         recycl_into_south = true;
10   }else
11      spawn(i+1,j));
12 }
13 if(recycle_into_east){
14   recycle_as_child_of();
15   j = j+1;
16   return this;
17 }else if(recycl_into_south){
18   recycle_as_child_of();
19   i=i+1;
20   return this;
21 }else
22   return NULL;
```

Figura 3.20: Optimización en TBB utilizando reciclado (TBB_v4)

En la línea 3 ponemos el flag`recycle_into_east` a verdadero si el vecino del este está preparado para ser ejecutado; en caso contrario, y si está preparado el vecino del sur, pondremos el flag `recycle_into_south` a verdadero (línea 9). Posteriormente, según qué flag este activado, nos reciclaremos en el vecino correspondientes en la línea 14 o línea 18 respectivamente. Es importante mencionar que en este ejemplo los datos son almacenados por filas. Si ambas tareas, este y sur, están listas, los datos de

cache pueden aprovecharse mejor si reciclamos en la tarea del este que es la que consumirá los datos de la misma fila. De este modo, el mismo thread/core que está ejecutando la tarea actual será el encargado de ejecutar la tarea vecina que utiliza los mismos datos ya almacenados en la cache, por tanto, aprovechará la localidad espacial. Así, en este caso, reciclamos en la tarea este y lanzamos la tarea sur como nueva tarea que será ejecutada posteriormente. En cualquier caso, el número de `spawns` en esta versión se reduce de $n \times n - 2n$ (en TBB_v1 y TBB_v2) a $n - 2$ (aproximadamente el tamaño de una columna).

Con intención de evaluar el impacto del mecanismo de reciclado y el aprovechamiento de la localidad espacial, hemos estudiando el rendimiento de dos nuevas versiones: TBB_v3, una versión que prioriza el reciclado en la tarea sur en vez de la vecina del este, que consigue reducir el número de spawns; pero no explotará la localidad espacial. Y la versión TBB_v4, que implementa el algoritmo descrito en la figura 3.20, de manera que aprovecha la localidad espacial reciclándose siempre si es posible en la tarea de la misma fila. Estos experimentos los hacemos para grano fino y medio, donde encontrábamos mayor problema de rendimiento (aproximadamente 200 y 400 FLOPS).

Por los resultados obtenidos (figura 3.21) vemos que TBB_v4 es claramente la mejor solución. De hecho, el speedup medido para otros tamaños de grano fino nos muestra que cuanto más fino es el grano más se notan las diferencias entre estas versiones. Además, es interesante señalar como una gran contribución la optimización introducida por el mecanismo de reciclado, viendo que la versión TBB_v3 mejora claramente la versión TBB_v1.

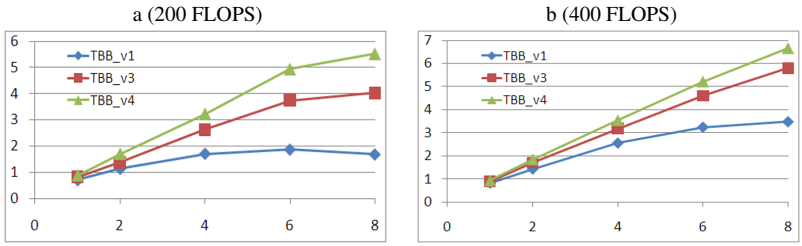



Figura 3.21: Speedup de las versiones optimizadas con grano de tarea constante para 1, 2, 4, 6 y 8 cores.

Para estudiarlas de manera más profunda, analizamos las tres implementaciones de TBB con la actividad *Call Graph* de Vtune, para entender lo que está ocurriendo con más detalle y las funciones que más tiempo están consumiendo. En la figura 3.22, mostramos la ratio del tiempo consumido por cada función respecto al total de la ejecución. En

concreto, mostramos aquellas funciones relacionadas con la creación y gestión de las tareas: (`spawn`(), `allocate`() y `get_task`()). El estudio para estas tres funciones es el mismo para TBB_v3 y TBB_v4. En esta figura señalamos los porcentajes de tiempo perdido por estas funciones los cuales son inferiores al 6 % . De hecho, podemos ver que para cada función interna la reducción es de un orden de magnitud respecto a la versión TBB_v1 (figura 3.15). Por ejemplo, una de las razones de esta reducción es que el número de llamadas de `spawn` en TBB_v1 era de 998000 (claramente una llamada por celda de la matriz), mientras que el número de llamadas en TBB_v3 y TBB_v4 es tan solo de 998.

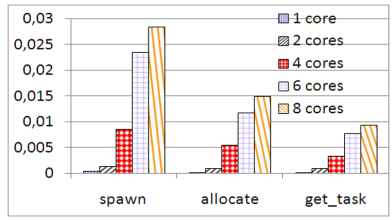



Figura 3.22: Ratio entre el tiempo empleado por las funciones que más tiempo consumen y el total de ejecución para las versiones optimizadas con grano de tarea constante.

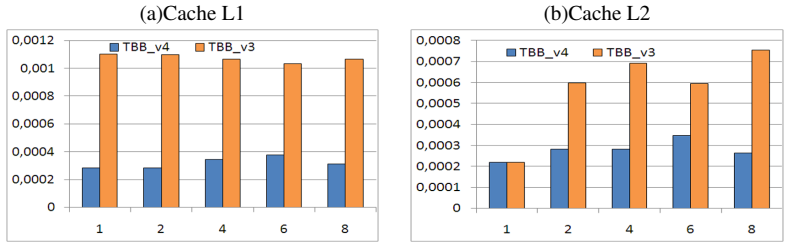



Figura 3.23: Ratio de fallos de datos en las cachés L1 y L2 en las versiones optimizadas en TBB con grano fino.

Por otro lado, gracias al muestreo de Vtune podemos recolectar el porcentaje de fallos en la cache L1 de datos y en la cache L2 de datos para TBB_v3 y TBB_v4. Las ratios de fallo de la figura 3.23 confirman que el rendimiento de la cache en TBB_v4 es superior como intuíamos antes de realizar los experimentos. Estudiando otras granularidades, vemos que el aprovechamiento de la cache es mayor cuando nos movemos a un grano más fino de tarea y se aprecia mayor diferencia en la cache de datos L1 que en la

cache de datos L2.

### 3.2.3. Tiling

Como hemos visto, el rendimiento de los algoritmos de wavefront decrece de manera consistente cuanto más fino es el grano de la tarea. La técnica de "tiling" es una solución bien conocida para obtener grano grueso que puede ser fácilmente aplicada a problemas wavefront. La idea es asignar un número igual de columnas y filas adyacentes a cada tarea. Esta técnica consigue dos objetivos: reducir el número de tareas de manera que se realicen menos `spawns` y ahorrar sobrecarga por disminuir el número de contadores, lo que implica menos colisiones además de aprovechar la localidad espacial de la cache con mayor intensidad. Ésta última opción se podría intensificar si los bloques son más rectangulares que cuadrados dando un mayor número de columnas que de filas al bloque. Así lo fallos de cache se verían disminuidos si existen estrategias de precarga en la cache.

Implementamos tiling cuadrado sobre TBB_v4 y medimos el speedup para diferentes tamaño de bloque de tareas en caso de grano fino. En nuestro experimentos encontramos que el speedup para 8 cores mejora alrededor de un 9 % con un tamaño de bloque BS = 10. Para bloques más grandes, el paralelismo decrece y por tanto, no obtenemos tanto rendimiento. A mayor tamaño de bloque, habrá menos bloques que puedan ejecutarse concurrentemente y de ahí que podamos obtener menos rendimiento.

En el capítulo 4 se aborda con mayor extensión la estrategia de tiling y se describen las propuestas con las que pretendemos automatizar la aplicación de esta optimización.

## 3.3. Plantilla TBB para problemas de tipo wavefront

Hemos explorado la aplicabilidad del modelo de programación basado en tareas para la paralelización del patrón de diseño wavefront. Encontramos que este patrón se ajusta bien con el paradigma debido a diversas razones:

- El actual estado del arte en cuanto a librerías basadas en tareas (OpenMP 3.0, TBB, CnC) proveen un modelo de programación en el cual se desarrollan expresiones de paralelismo utilizando tareas, dejando a un lado la planificación de éstas al "runtime" de las librerías, ofreciendo un entorno de programación más productivo.
- Las tareas son mucho más ligeras que los threads, los cuales permiten una implementación paralela más escalable de problemas de wavefront reales (la carga computacional de cada celda tiende a ser más pequeña).

- El planificador de tareas en estas librerías está basado en la estrategia de planificación work-stealing, logrando una mejor distribución del balanceo de carga que la que ofrece el planificador de threads del sistema operativo, mejorando así la escalabilidad.

En los experimentos ya comentados, evaluamos las funcionalidades que cada librería provee para que el usuario las utilice cuando implementa problemas de tipo wavefront, concluyendo que TBB ofrece algunas ventajas que permiten implementaciones más eficientes del patrón, como puede ser la captura atómica de variables compartidas para la sincronización de tareas y el mecanismo de paso de tareas (o reciclado de tareas), el cual es esencial para reducir el overhead de creación y planificación de tareas. Sin embargo, para los programadores que no son expertos en programación paralela, puede ser difícil implementar un algoritmo wavefront en un sistema multicore; ya que hay que controlar la gestión de tareas, sincronizarlas, saber aprovechar el balanceado de carga y lograr una correcta utilización del mecanismo de reciclado. Para aliviar estas dificultades, proponemos un template o plantilla de alto nivel en el cual el programador sólo tendrá que facilitar un patrón de dependencias y la computación de cada tarea. La plantilla está construida sobre TBB en un estilo similar al resto de las plantillas de TBB ya estudiadas en el capítulo anterior, como pueden ser `parallel_for` o `pipeline`.

El principal objetivo aquí perseguido es describir esta plantilla, su uso y ventajas de programabilidad. Además, realizaremos una serie de experimentos con problemas reales, como el codificador H264, para evaluar el rendimiento, el overhead introducido por la plantilla y las ganancias en términos de programabilidad.

### 3.3.1. Plantilla wavefront

El objetivo de la plantilla es minimizar el tiempo y el esfuerzo que el usuario necesita invertir para desarrollar un algoritmo wavefront en paralelo. Para poder realizar esto, la plantilla le requiere al programador una información mínima para automatizar y generar un código similar al esquema básico de problemas wavefront que vimos en 3.1, y mostrado en la figura 3.3.

En la figura 3.24 se muestra un esquema que explica como se utiliza la plantilla. Al programador se le solicita que facilite la descripción del patrón de dependencias, para lo que deberá escribir un archivo llamado fichero de definición como el que vemos en la figura 3.25. En ésta podemos ver el fichero de definición del ejemplo del problema básico de wavefront 2D presentado en 3.1 y a su derecha la descripción gráfica del patrón de dependencias y la matriz de contadores. El programador también deberá escribir un fichero cabecera `userMethods.h`, donde implementará el método `executeTask`,

el cual especifica el trabajo que realiza cada tarea. En este mismo fichero también hay que programar el método `dataInit`.

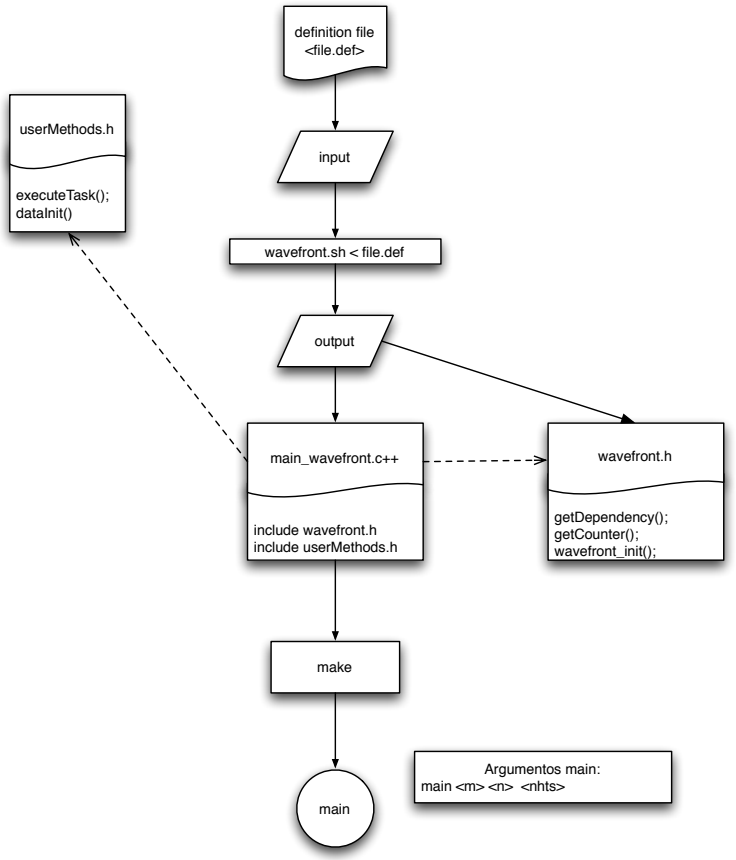



Figura 3.24: Esquema de utilización de la plantilla.

En primer lugar, el fichero de definición es pre-procesado por nuestro generador de código, `gen`, llamado desde el script `wavefront.sh`. Este comando devuelve una cabecera, `wavefront.h`, (que contiene las funciones y métodos que implementan el patrón de dependencias) y que es incluido en el fichero principal `main_wavefront.c++` y que resulta también como salida de la llamada al script. El fichero principal también incluye la cabecera implementada por el usuario `userMethods.h`. Tras compilar este programa obtenemos un ejecutable con los siguientes parámetros de entrada requeridos:

el tamaño del problema y el número de threads que queremos utilizar.

El fichero de definición tiene 5 secciones como muestra la figura 3.25. Las secciones 1, 2, 4 y 5 se utilizan para especificar dominios o regiones en el cual los índices son válidos. Una región es un conjunto de índices rectangulares dentro de un rango determinado. En particular, nosotros utilizamos la notación LHS de `F90-Matlab` para definir una región. Cada dimensión de una región es una tupla de tres elementos expresados de la forma LHS, donde l y h representan el inicio y el fin de un vector y s el paso. Una región de "d" dimensiones se expresa de la forma [$l_1$:$h_1$:$s_1$,...,$l_d$:$h_d$:$s_d$]. Si existe ausencia de paso, por defecto se toma el paso 1. Si en una de las dimensiones tan solo tenemos un índice concreto, éste se expresará como un número (Ej: [1,1:n]). Cuando la dimensión es indicada con ":" representa a todo el dominio de ese índice. Una expresión del tipo !(k) significa "todos los elementos excepto cuando el índice tenga valor k".

```
1 //Sección 1: Data space
2 [0:n-1,0:n-1]
3 //Sección 2: Task space
4 [1:n-1, 1:n-1]
5 //Sección 3: Índices
6 <i, j>
7 //Sección 4: Vectores de
      dependencias
8 [1:n-1 , 1:n-1 ] -> (0,1); (1, 0)
9 //Sección 5 (Opcional): valor
      contadores
10 [1,1] = 0
11 [1, 2:n-1 ] = 1
12 [2:n-1 , 1] = 1
13 [2:n-1, 2:n-1 ] = 2
```

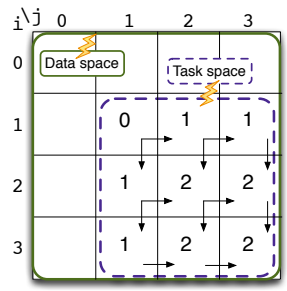



Figura 3.25: Fichero de definición para el problema simple en 2D.

La primera sección del fichero de definición, línea 2, señala el dominio de todo el grid, mientras que la segunda, línea 4, indica el dominio espacial de las tareas. En la línea 6, asociamos "i" como primera variable para el índice de la primera dimensión y "j" para el de la segunda. Esas variables no son necesarias, en particular, en este archivo de definición; pero en algún ejemplo, que veremos a continuación si lo serán para indicar dominios de regiones parametrizados y para especificar la información de los contadores. En la cuarta sección, en la línea 8, representamos la información de dependencias. Esta información se especifica con una región que sigue la estructura LHS y una lista de dependencias separadas por ";" en LHS. La línea 8 captura las dependencias horizontales y verticales en la figura 3.2(a): indicando que en la región [1:n-1,1:n-1] se aplican los

vectores (0,1) y (1,0). Las dependencias horizontales de la última fila, y las dependencias verticales de la última columna (es decir, las fronteras) son casos particulares de la dependencia general de la línea 8 y la plantilla las maneja automáticamente (son dependencias fronteras de región que se detectan internamente). Es muy importante asegurar que las regiones no se solapen y que el vector de dependencias represente una dirección relativa o desplazamiento desde cada celda de la correspondiente región. Además, el programador puede proporcionar información acerca de la localidad cuando hay una lista de dependencias, exponiendo primero la de mayor prioridad, como expresamos en la línea 8, poniendo primero la dependencia asociada al recorrido de las filas.

Desde la información de dependencias es posible generar automáticamente el valor de los contadores, pero como veremos más adelante, en algunos casos, se puede proporcionar una inicialización más rápida dando el valor directo de estos contadores como podemos ver en la última sección del fichero de definición. Primero, tendremos un identificador de región LHS, pero ahora esa región LHS contiene el valor del contador para cada entrada de la región.

En la figura 3.26 podemos ver la notación BNF para la sintaxis del fichero de definición. La primera regla en la línea 1 señala que el fichero tendrá un *Data Grid* (primera sección), un *Task Grid* (segunda sección), una línea *Index* (tercera sección), una o más regiones de dependencias (*RegionDep* ) (cuarta sección) y cero o más *Counters Values* (quinta sección). El *Data Grid* y el *Task Grid* son definidos en las líneas 6 y 7 como una *Region*. Una *Region* es un conjunto de celdas de un Grid. Podemos definir una *Region* en notación BNF como $n$ *Vectors* separados por "," como muestra la línea 10 de la figura. Un *Vector* es una expresión (*exp*) o dos o tres expresiones separadas por ":" (línea 14). Si definimos un *Vector* como una expresión simple esto significa que es una expresión aritmética (un número, operación, etc.). Sin embargo, tres expresiones separadas por ":" significa un vector del tipo l:h:s. *Index* es una etiqueta para nombrar variables que la plantilla usa para cada dimensión de la matriz. Así que las variables son letras (en la línea 16), separadas por ",". A su vez, *exp* es una operación aritmética, un número, incluyendo variables o índices que son representados por *Digit* (digitos), *Letters* (letras) o *sign* (símbolos o signos).

Podemos escribir en el fichero de definición una o más dependencias que son definidas en la línea 9. Estos vectores de dependencias están formados por una *Region* seguido de "−>" y un conjunto de dependencias (*Dep* en la línea 11 de la figura) separadas por ";". Una dependencia (*Dep*) se construye con un *Pair* o un *Vector_dep*. Un *Pair* son dos o más expresiones entre "(" y ")", separadas por ",", una expresión para cada dimensión de la matriz; y un *Vector_dep* es al menos un *Vector* en una de las dimensiones. Desde aquí en adelante llamaremos dependencia simple a un *Pair* y vector de dependencias a un *Vector_dep*. Finalmente definimos los contadores utilizando la notación "=" para asignar una expresión a una *Region*.

```
<Configurator> :: = <Datagrid>
                     <TaskGrid>
                     <Index>
                    +(<RegionDep>)
                    *(<Counters>)
<Datagrid> :: = <Region>
<TaskGrid> :: = <Region>
<Counters> :: =  <Region> = <exp>
<RegionDep> :: = <Region> -> <Dep>
<Region> :: = [<Vector>*(,<Vector>)] \\
<Dep> :: = <Dep>; <Dep>  ; | <Pair>| <Vector_dep>
<Vector_dep> :: = (<Vector>*(, <exp>)) |
                      (*(<exp>,) <Vector>)
<Vector> :: = <exp> | <exp> : <exp> |
              <exp>:<exp>:<exp> | :
<Index> :: = < +<Letter> *(, + <Letter>) >
<Pair>  :: = (<exp>*(,<exp>))
<exp>   :: = +<Digit> | +<Letter>|
             +<exp> <sign> <exp>
<Letter> :: = a|b|c|...|y|z
<Digit> :: = 0|1|2|3...|9
<Sign> :: = *|+|-|/|%
```

Figura 3.26: Sintaxis del fichero de definición para wavefront en notación BNF.

La sección de dependencias es la más importante del fichero de definición. En las líneas siguientes presentamos unos conceptos importantes sobre las dependencias. Hemos definido la sección de dependencias utilizando regiones y dependencias. Sin perder generalidad, trabajaremos a partir de aquí siempre con una matriz 2D y paso igual a uno (así que lo omitiremos de los vectores). Siendo los resultados extrapolables a matrices de cualquier dimensión.

Necesitamos conocer como obtener las tareas dependientes de un elemento (i,j) aplicando sus dependencias. Así que lo primero que hay que averiguar es a qué región pertenece dicho elemento para poder acceder a sus dependencias.

Decimos que un elemento pertenece a una región en los siguientes casos:

Dado $X = (i, j)$ perteneciente a la región R $[lx : hx, ly : hy]$, significa que el elemento (i,j) está dentro de los límites de la región: $lx \leq i \leq hx$ , y $ly \leq j \leq hy$.

Dado $X = (i, j)$ perteneciente a la región R $[x, ly : hy]$ significa $i = x$ y $ly \leq j \leq hy$.

Dado $X = (i, j)$ perteneciente a la región R $[lx : hx, y]$ significa $lx \leq i \leq hx$, y $j = y$.

Dado $X = (i, j)$ perteneciente a la región R $[x, y]$ significa $i = x$ y $j = y$.

Hemos visto que hay dos tipos de dependencias. Las dependencias simples y los vectores de dependencias. Las dependencias simples son fáciles de aplicar ya que son un par de expresiones. Por ejemplo, dado el elemento $X = (i, j)$ perteneciente a la región R con la siguiente dependencia $R$ -> $(x, y)$. Para obtener el nuevo elemento que depende de X, tan sólo tenemos que sumarle a la coordenada de cada dimensión del elemento el correspondiente desplazamiento que indica la dependencia. El resultado es el nuevo elemento: $(i, j) + (x, y) = (i + x, j + y)$.

Aplicar un vector de dependencias es más complicado. Un vector de dependencias expresa, generalmente, más de una dependencia. Podríamos desarrollar el vector de dependencias iterando sobre el vector $(dlx : dhx, dly : dhy)$ creando dependencias simples y aplicarle cada una de ellas al elemento, tal y como acabamos de ver. Por ejemplo, si tenemos el vector $(1 : 3, 0)$, este se transforma en las dependencias simples (1,0), (2,0) y (3,0), las cuales pueden ser aplicadas fácilmente a un elemento $X = (i, j)$. En cada iteración del bucle añadimos cada tarea dependiente obtenida al conjunto de tareas dependientes del elemento X, como vemos en el pseudo código de la figura 3.27.

```
1 // Dado el vector dlx:dhx;dly:dhy desarrollamos el vector de dependencias
2 // para obtener dependencias simples
3 for(y=dly; y<=dhy; y++){
4  for(x=dlx; x<=dhx; x++){
5   dependent_task->add((i,j) + new simple_dependency(x,y));
6  }
7 }
```

Figura 3.27: Pseudocódigo para desarrollar un vector de dependencias.

Dada una región $R = [lx : hx, ly : hy]$ y una de sus dependencias simples $(dx, dy)$, podemos obtener la región R2, que define el área donde están las tareas dependientes de la región añadiendo las dependencias simples a la región R. Todas las tareas pertenecientes a la nueva región R2 dependen de alguna tarea que pertenece a R.

$$R2 = [lx : hx; ly : hy] + (dx, dy) = [lx + dx : hx + dx, ly + dy : hy + dy] \quad (3.1)$$

Dado $X = (i, j)$ y un vector de dependencias $(dlx : dhx, dly : dhy)$, podemos obtener una región con las tareas dependientes de X añadiendo el vector de dependencias al elemento X utilizando la siguiente ecuación:

$$(i,j) + (dlx : dhx, dly : dhy) = [i + dlx : i + dhx, j + dly : j + dhy] \tag{3.2}$$

Dado $X = (i,j)$ y una región con elementos dependientes de X, $R = [dlx : dhx, dly : dhy]$, podemos obtener el vector de dependencias que aplicamos a X para obtener R utilizando la regla inversa:

$$[lx : hx, ly : hy] - (i,j) = (lx - i : hx - i, ly - j : hy - j) \tag{3.3}$$

La distancia entre dos elementos $A = (x,y)$ y $B = (i,j)$ se calcula restando las coordenadas de cada dimensión de los elementos. Así, definimos la distancia entre A y B por $D(A,B) = (i - x, j - y)$.

Dado $X = (i,j)$ perteneciente a la región R con las dependencias $R$ -> $d_1, d_2, d_3..., d_n$. Podemos decir que hay un ciclo cuando aplicando dos o más dependencias, una o más veces cada una de ellas, obtenemos nuevamente el elemento X.

La dependencia $d_e = (0,0)$ es el elemento neutro de la operación aplicación entre elementos y dependencias. Esto significa que aplicando esta dependencia a un elemento siempre obtenemos el mismo elemento.

Dado un conjunto de dependencias $d_1, d_2, d_3, \cdots, d_n$, llamamos combinación lineal de dependencias a la siguiente ecuación:

$X_1 * d_1 + X_2 * d_2 + \cdots + X_n * d_n = d_m$ donde $X_i$ son enteros mayores o iguales que cero y $X1 + X2 + \cdots + X_n > 0$. De modo que obtenemos una nueva dependencia $d_m$. Esta dependencia es equivalente a añadir $X_i$ veces cada dependencia. Si $d_m$ es el elemento neutro y tenemos al menos dos $X_i$ mayores que cero podemos garantizar que la aplicación de las dependencias $d_1, d_2, d_3..., d_n$ da lugar a un ciclo.

Dado $X = (i,j)$, $Y = (i2, j2)$ pertenecientes a la región $R$, en la que están definidas las dependencias $d_1, d_2, d_3, \cdots, d_n$. Decimos que Y depende directamente de X si añadiendo algún $d_i$ a X obtenemos Y.

Dado $X = (i,j)$, $Y = (i2, j2)$ y $Z = (i3, j3)$ pertenecientes a la región con dependencias $R$ -> $d_1, d_2, d_3, \cdots, d_n$., donde Y es dependiente de X y Z dependiente de Y. Entonces, podemos decir que Z depende de X por la propiedad transitiva.

Con la información proporcionada por el fichero de definición, utilizando el generador de código que mencionamos anteriormente, se genera un archivo de cabecera *wavefront.h*, que contiene todas las clases y métodos necesarios para usar la plantilla. Así que

el programador, tras escribir el fichero de definición, debe invocar a este generador de código e incluir la cabecera resultante en su módulo principal, como vemos en la figura 3.29 línea 1.

```
1
2 void Operation::executeTask()
3 {
4   int i = GetFirst();
5   int j = GetSecond();
6   A[i][j]=foo(gs, A[i][j], A[i-1][j], A[i][j-1]));
7 }
8
9 int dataInit (int argc, char **argv)
10 {
11   //Procesado de argumentos
12    A = malloc(size);
13 }
```

Figura 3.28: Fichero userMethods.h: computación de una tarea e inicialización del problema.

```
1 #include "wavefront.h"
2 #include "userMethods.h"
3 int main(){
4   ....
5   wavefront_init(); // Inicializar TBB y variables de la plantilla
6   wavefront->run(); // Ejecutar código wavefront
7   ....
8 }
```

Figura 3.29: Función principal para ejecutar un código wavefront.

En la función `main`, se llama a la función `wavefront_init()` y al método `run()` (líneas 5 y 6). Básicamente, en la primera llamada se invocan las inicializaciones de las rutinas de TBB y algunas variables necesarias para la plantilla y para inicializar la matriz de contadores. El método `run`, definido en `wavefront.h`, es llamado para lanzar las tareas iniciales (aquellas con el contador a 0).

Otra tarea que el programador tiene que llevar a cabo, es sobrescribir el método `Operation::executeTask()` en el fichero `userMethods.h`, dónde se especifica la operación que tiene que ser computada por cada celda. `GetFirst`() (línea 4) y `GetSecond`() (línea 5) son métodos de la plantilla que proporcionan las coordenadas de cada celda, las cuales son imprescindibles para cada computación.

Resumiendo, los pasos que debe dar un programador para usar esta nueva plantilla de TBB son:

- Confeccionar un fichero de definición como el de la figura 3.25.
- Ejecutar el generador de código con este fichero de definición como entrada. Al final de este paso obtendremos un fichero `wavefront.h` y el programa principal `main_wavefront.c++` (fichero definido en la figura 3.29).
- Codificar la cabecera `userMethods.h` donde se definen la computación de una tarea y la inicialización del problema (figura 3.28).

### 3.3.2. Detalles de implementación

A continuación pasamos a describir los detalles de implementación del generador de código y del fichero `wavefront.h` que se genera. La plantilla ha sido implementada para soportar problemas que tengan un patrón wavefront 2D y 3D. En este escenario, y sin pérdida de generalidad, todas las tareas internamente son identificadas con un entero, ID, empezando desde el 0. El fichero generado `wavefront.h` contendrá las clases y métodos representados en el diagrama de clases de la figura 3.30. En el podemos ver que hay tres clases, la clase `Wave<Operation>` que es una clase parametrizada con la clase Operation.

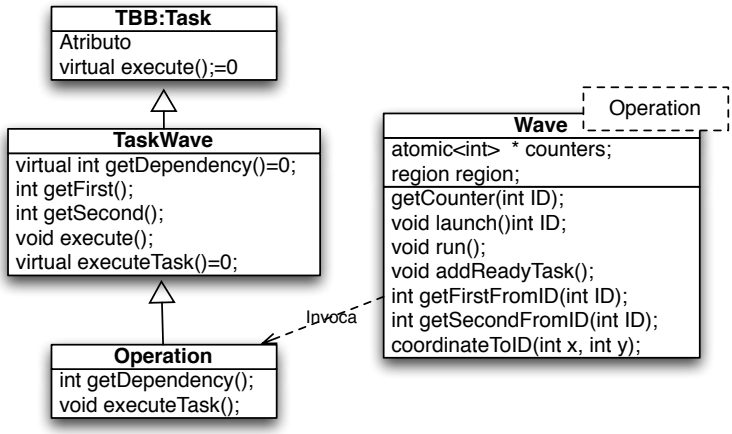



Figura 3.30: Diagrama de clases de la plantilla.

La clase Operation es una clase que hereda de `TaskWave`, y ésta hereda a su vez de `TBB:task`. Por tanto, Operation es una clase que implementa una tarea de TBB, mientras que la clase `Wave` es la encargada de almacenar toda la información del problema: matriz de contadores, métodos necesarios para actualizar las dependencias, y si es necesario realizar el spawn o reciclar en la siguiente tarea. Dentro de la clase `Wave` destacamos los siguientes métodos que explicaremos a continuación: `GetCounter(int ID)`, `launch(int ID)`, `run()`, `addReadyTask()`. Por otro lado en la clase `TaskWave` destacan: `GetFirst()`, `GetSecond()`, `GetDependency()` y `executeTask()`. Por último la clase `Operation` que hereda de la clase `TaskWave`, tiene que sobrescribir los métodos declarados como virtuales en la clase `TaskWave`. Los métodos `GetFirst()`, `GetSecond()` y `GetThird()` pueden ser utilizadas para obtener las coordenadas (son necesarias para accedes a los datos de la matriz espacial). Hay también métodos como `GetFirstFromID`, `GetSecondFromID` y `GetThirdFromID` (definidas en la clase `Wave`), que pasándole el ID por parámetro (un entero que identifica una tarea), devuelven cada una de las coordenadas asociadas a esa tarea. El método inverso también está implementado: `CoordinateToID(i,j,k)`, que devuelven el ID correspondiente a las coordenadas pasadas como parámetros. Estos métodos están implementadas para 2D y 3D, respectivamente. El método `GetCounter(ID)` se genera a partir del fichero de definición y se define en la cabecera `wavefront.h`. La maquinaria interna de la plantilla wavefront está basada sobre la clase `Wave`, la clase `TaskWave` y la clase `Operation` (que puede ser utilizada directamente por el usuario). Esta última clase tiene dos métodos públicos: `executeTask()`, que tiene que ser sobrescrito por el usuario (como vemos para el problema básico básico 2D en la figura 3.29) y `GetDependency(o)`. El último método también es generado automáticamente desde el fichero de definición y están incluido en la cabecera `wavefront.h`; aunque posteriormente puede ser modificado por el programador para realizar una optimización del código. En el fichero `wavefront.h` también está implementada la función `wavefront_init()`, la cual inicializa algunas variables (las dimensiones de la matriz, las fronteras del espacio de tareas) e inicializa el objeto `Wave<Operation> * wavefront`. En el constructor de este objeto, se reserva el espacio para la matriz de contadores y estos se inicializan mediante un `parallel_for`. Este bucle itera sobre los elementos de la matriz y llama al método `GetCounter(ID)` por cada uno, tal y como vemos en el pseudocódigo del cuerpo del bucle entre las líneas 1 y 5 de la figura 3.31. También recolectamos una lista de las tareas con valor de contador igual a 0, que serán las primeras en ejecutarse pues no tienen dependencias de ninguna tarea (en la línea 3 de la figura se añaden tareas a una lista de tareas preparadas). El método `GetCounter`, generado desde el fichero de definición del problema básico de wavefront 2D, se muestra en la figura 3.32. Podemos ver, que devuelve un valor del contador dependiendo de la coordenada del ID de la tarea pasada por parámetro.

```
1 template<class Operation>  int Wave<Operation>::void body(int ID){
2     if((counter[ID]  = GetCounter(ID))==0){
3       addReadyTask(new Operation(ID));
4     }
5 }
6
7 template<class Operation>  int Wave<Operation>::void run(){
8   spawn_root_and_wait(readyTaskList);
9 }
```

Figura 3.31: Pseudocódigo del cuerpo del `parallel_for` para inicializar los contadores así como del método `run()`.

```
1 template<class Operation>  int Wave<Operation>::GetCounter(int ID){
2   int i= GetFirstFromID(ID);
3   int j= GetSecondFromID(ID);
4   int counter = 0;
5   if((i==1) && (j==1))
6     counter= 0;
7   if((i==1) && (j>=2 && j<=m-1))
8     counter= 1;
9   if((i>=2 && i<=m-1) && (j==1))
10    counter= 1;
11  if((i>=2 && i<=m-1) && (j>=2 && j<=m-1))
12    counter= 2;
13  return counter;
14 }
```

Figura 3.32: Método `GetCounter()` para el problema básico 2D. Este método devuelve el número de tareas de las que depende la tarea ID.

Si no se indicó la sección 5 en el fichero de definición, la inicialización de los contadores es distinta. Se proporciona un método `GetCounter` por defecto, como el que podemos ver en la figura 3.33. En él, podemos ver que para cada tarea se llama al método `GetDependency` tantas veces como dependencias tenga (como explicaremos más adelante cuando se detalle el método `launch`()) y se incrementa el valor del contador de la tarea dependiente. Al final de computar todas las tareas, los contadores están inicializados. También varía el método `body` de la figura 3.31. Ya que en este caso no tiene que realizar ninguna asignación como se realiza en la línea 2. Símplemente cada elemento llama al método `GetCounter`. Tras inicializar todos los contadores se realiza otra iteración para ver cuales tienen aún el contador a 0, estos serán los contadores de las tareas preparadas. Como se puede intuir, esta forma de inicializar los contadores es mucho más lenta como explicaremos más adelante en los resultados experimentales. Una vez

sobrescrito `executeTask()` y compilado el problema, podemos utilizar la plantilla.

```
template<class Operation>  int Wave<Operation>::GetCounter(int ID){
 TaskWave * t = (TaskWave*) t1;
 int IDRecycledTask = -1;
 int o=1;
 int IDdepTask= t->getDependency(o++);
 while(IDdepTask!=NO_MORE_DEPENDENCIES){
  if ((IDdepTask!=INVALID_POSITION) &&(--counters[IDdepTask]==0)){
   counter[IDdepTask]++;
   }
  }
  IDdepTask = t->getDependency(o++);
 }

 return -1;
}
```

Figura 3.33: El método `GetCounter()` por defecto si se omite la sección de inicialización de contadores.

En la figura 3.34 podemos ver como interaccionan las clases descritas anteriormente en el proceso de ejecución.

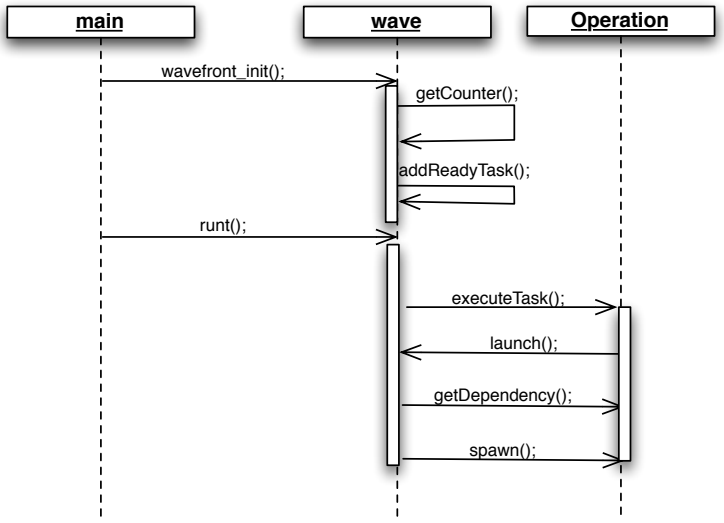



Figura 3.34: Diagrama interacción entre las clases de la plantilla.

En primer lugar inicializamos el problema llamando a `wavefront_init()`. Tras esta llamada, podemos llamar al método `run()` (el método se muestra a partir de la línea 7 de la figura 3.31) justo después de haber creado el objeto wavefront como vimos en la figura 3.29. Este método `run()` realiza un spawn de la lista de tareas preparadas. Cada tarea que se ejecuta, primero invoca el método `executeTask()` y tras su ejecución el mecanismo interno llama al método `launch`. Este método es el encargado de identificar todas las tareas dependientes (llamando a `GetDependency`), decrementar sus contadores y crear las nuevas tareas o reciclarse en alguna de ellas si están preparadas. Este proceso itera hasta que no quedan tareas dependientes de la tarea actual.

```
1 template<class Operation>  int Wave<Operation>::launch(task *t1){
2  TaskWave * t = (TaskWave*) t1;
3  int IDRecycledTask = -1;
4  int o=1;
5  int IDdepTask= t->getDependency(o++);
6  while(IDdepTask!=NO_MORE_DEPENDENCIES){
7   if ((IDdepTask!=INVALID_POSITION) && (--counters[IDdepTask]==0)){
8    if (IDRecycledTask>=0){
9     t->spawn( *new( this->allocate(t)  ) TaskWave(this,IDdepTask) );
10   }else{  // no recycled task yet
11    t->recycle_as_child_of(*t->parent());
12    IDRecycledTask = IDdepTask;
13   }
14  }
15  IDdepTask = t->getDependency(o++);
16 }
17 if (IDRecycledTask>0)
18  t->ID = IDRecycledTask;
19 return IDRecycledTask;
20 }
```

Figura 3.35: El método `launch()` comprueba los contadores y lanza o recicla las nuevas tareas.

En la figura 3.35 mostramos el kernel de la maquinaria de la plantilla que está implementado en el método `launch`. Señalar que este método es llamado por cada tarea justo después de ejecutar el cuerpo de la misma, `executeTask()`. Así que el método `launch()` tiene que decrementar los contadores de todas las tareas dependientes y lanzarlas si están preparadas. En la línea 5 primero obtenemos el ID de la primera tarea dependiente. `IDdepTask`, desde el método `GetDependency()`. Entonces, el bucle while de la línea 6 irá visitando todas las tareas dependientes hasta que `GetDependency` devuelva `NO_MORE_DEPENDENCIES`. Para cada una de esas tareas dependientes, primero en la línea 6, comprobamos que el $ID$ de la tarea dependiente

es válido (IDdepTask!=NO_MORE_DEPENDENCIES, debido a una posición inválida (INVALID_POSITION, podría ser un número negativo) y en esos casos decrementamos el valor del contador de la tarea y comprobamos si después es nulo. En tal caso, si todavía no hemos decidido reciclarnos en una tarea dependiente previa (línea 7), optamos por reciclarnos en la primera que es nula, anotando esta tarea como tarea en la que nos reciclaremos (línea 8); en caso contrario realizaremos un spawn de la nueva tarea. Hay que señalar que una tarea se va a reciclar en la primera tarea que esté preparada. Así que el orden en el cual GetDependency devuelve las tareas dependientes es relevante. Esto es, el mecanismo para lanzar tareas está basado en reciclar la primera tarea devuelta por GetDependency() y realizar spawns de las siguientes. Además, las tareas dependientes pueden explotar mejor la localidad espacial de la cache de la tarea actual si GetDepedency devuelve la tarea adecuada primero. Esto puede ser conseguido adecuando el orden de las dependencias en el fichero de definición. Finalmente, en la línea 18, en el caso de que la tarea actual vaya a ser reciclada en otra, actualizamos el $ID$ de la tarea por la nueva en la cual va a ser convertida.

A pesar de estar la plantilla diseñada para generar los métodos necesarios utilizando el fichero de definición, el programador, siempre puede escribir los métodos GetDependency y GetCounter de manera manual o modificar los ya generados.

La identificación de las tareas dependientes son llevadas a cabo por iteración llamando al método GetDependecy(o++) como acabamos de ver. En la figura 3.36 mostramos GetDependency(o) generado automáticamente para el problema básico wavefront 2D. Como vemos, este método primero identifica las coordenadas (i, j) de llamada a la tarea. Entonces, dependiendo de la región en la que esté la tarea localizada, los correspondientes vectores especificados en el fichero de definición se aplicarán uno a uno. Por ejemplo, en el problema básico de wavefront 2D definimos una región (ver figura 3.25) en la que aparecen dos dependencias. Entonces, asumiendo que una tarea está en la primera región, cada vez que esta tarea llama al método GetDependency(o++), un vector de dependencia es aplicado: (0,1) la primera vez que (o==1) y (1,0) la segunda vez con (o==2). Para cada vector, las coordenadas de tareas resultantes son validadas si están dentro del espacio de tareas. En caso positivo el ID de la tarea resultante es devuelto. En otro caso se devolverá una constante INVALIDAD_POSITION señalando que la tarea está fuera de las fronteras y que no hay más tareas en esa dirección. Cuando no hay más vectores de dependencias en la región se devolverá la constante NO_MORE_DEPENDENCIES, así que el método launch() se dará cuenta de que todas las tareas dependientes han sido consideradas y lanzadas en caso de haber estado preparadas.

A continuación vamos a explicar cuales son los pasos que realiza el generador de código. El generador de código es un programa secuencial que realiza un preprocesado del fichero de definición para generar los métodos necesarias que acabamos de ver.

Como estudiamos en la sección previa, el fichero de definición tiene cinco secciones. Las secciones 1 y 2 las utilizaremos para crear el método `wavefront_init()` que se incluye el fichero `wavefront.h`. Este método construye el objeto wavefront necesario para ejecutar el problema, utilizando los límites de las regiones que hemos introducido en las secciones 1 y 2 del fichero de definición tal y como vemos en la figura 3.37.

```
1 int Operation::GetDependency(int o){
2   int IDdepTask = NO_MORE_DEPENDENCIES;
3   int i, j, i1, j1;
4   i = getFirst();
5   j = getSecond();
6   if (Region1(i,j)){
7     if (o==1){
8       i1 = i + 0;  j1 = j + 1;
9       bool ok = CheckTaskCoordinates(i1,j1);
10      if (!ok) IDdepTask = INVALID_POSITION;
11      else IDdepTask = CoordinateToID(i1,j1);
12    }
13    if (o==2){
14      i1 = i + 1; j1 = j + 0;
15      bool ok = CheckTaskCoordinates(i1,j1);
16      if (!ok) IDdepTask = INVALID_POSITION;
17      else IDdepTask = CoordinateToID(i1,j1);
18    }
19  }
20  return IDdepTask;
21 }
```

Figura 3.36: Método `GetDependency()` generado para el problema básico 2D desde su fichero de definición.

```
1 Wave<Operation> wave;
2 wavefront_init(){
3   wave = new Wave<Operation>(0, n-1, 0, n-1);
4 }
```

Figura 3.37: Método `wavefront_init()` generado para el problema básico 2D desde su fichero de definición.

La sección 3, la utiliza el fichero de definición para referirse a los índices cuando sea necesario. La sección 4 es utilizada para generar el método `GetDependency`. Primero utiliza la región para crear las condiciones de la figura 3.36 y posteriormente las dependencias para realizar el desplazamiento. Por último la sección 5 es utilizada para generar

el método `GetCounter`. Nuevamente las regiones se usan para generar las condiciones y el valor del contador para realizar la asignación correspondiente.

### 3.3.3. Aplicación a problemas reales

A continuación presentamos cinco problemas reales con diferente patrón de dependencias, los cuales consideramos que son representativos de la mayoría de problemas que podemos encontrar.

#### 3.3.3.1. Algoritmo Smith-Waterman

Los problema de alineamiento de secuencia se han convertido en una herramienta indispensable para la biología molecular, existiendo hoy en día un número alto de algoritmos de alineamiento de secuencia muy variados y con distinto grado de complejidad.

```
1 for (i=1; i<m; i++){
2   for (j=1; j<m; j++){
3     temp[0] = A[i-1][j-1]+similarity_score(L[i-1], K[j-1]);
4     temp[1] = A[i-1][j]-delta;
5     temp[2] = A[i][j-1]-delta;
6     temp[3] = 0;
7     A[i][j] = find_array_max(temp, 4);
8   }
9 }
```

Figura 3.38: Código secuencial del Smith-Waterman: delta es una constante.

El algoritmo Smith-Waterman [16] es utilizado para calcular la distancia entre dos secuencias típicas de ADN o proteínas utilizando programación dinámica. Para ello es necesario la creación de una matriz, de $m \times m$ celdas que indiquen el coste para cambiar una subsecuencia de una de las cadenas por otra subsecuencia de la otra cadena.

El algoritmo recorre una matriz por filas como podemos ver en el código secuencial representado en la figura 3.38 donde en cada iteración realiza el cómputo de la ecuación

$$A(i,j) = Max(A(i-1,j-1)+s(i-1,j-1), A(i-1,j)-delta, A(i,j-1)-delta) \tag{3.4}$$

, donde s es una función de cálculo de similitud.

Por otro lado, la implementación paralela del algoritmo Smith-Waterman se realiza computando las celdas de las antidiagonales en paralelo. Vemos en la ecuación 3.4 que de cada celda de la matriz dependen los vecinos a distancia (1,0), (1,1) y (0,1), respectivamente. Sin embargo, la dependencia (1,1) está implícita en las dependencias (1,0) y (0,1) por lo que no es necesario tenerla en cuenta. Por tanto, podemos decir que el Smith-Waterman es un problema de tipo wavefront como indican los autores en el trabajo [4]. El patrón de dependencias de este problema coincide con el problema básico 2D que estudiamos en este capítulo. Por tanto, podríamos expresar el fichero de definición para implementarlo con nuestra plantilla de una manera idéntica tal y como vemos en la figura 3.39. También es necesario implementar el método `executeTask` y `dataInit` (incluidos dentro del fichero `userMethods.h`) que están representados para el Smith-Waterman en la figura 3.40 utilizando las funciones auxiliares (definidas en las figuras 3.41 y 3.42) y la figura 3.43 respectivamente.

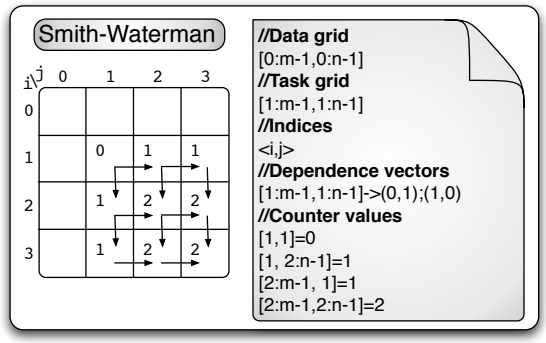



Figura 3.39: Fichero de definición del problema Smith-Waterman y patrón de dependencias.

```
1 void Operation::executeTask() {
2  int i  = GetFirst();
3  int j  = GetSecond();
4  temp[0] = A[i-1][j-1]+similarity_score(L[i-1],K[j-1]);
5  temp[1] = A[i-1][j]-delta;
6  temp[2] = A[i][j-1]-delta;
7  temp[3] = 0;
8  A[i][j] = find_array_max(temp,4);
9 }
```

Figura 3.40: Método `executeTask()` del Smith-Waterman.

```
double similarity_score(char a, char b){
  if(a==b) return 1.;
  else return -mu;
}
```

Figura 3.41: Función `Similarity_score()`. Compara dos secuencias de caracteres: `mu` es una constante.

```
double find_array_max(double array[], int length){
  double max = array[0];
  for(int i = 1; i<length; i++){
    if(array[i] > max){
      max = array[i];
     }
  }
  return max;
}
```

Figura 3.42: Función `find_array_max()` encuentra el valor más grande de un array.

```
int dataInit (int argc, char **argv)
{
//Procesado de argumentos
 ....
//Lectura de las secuencias L y K
 ....
 A = malloc(size);
}
```

Figura 3.43: Función `dataInit()` del Smith-Waterman.

#### 3.3.3.2. Problema Checkerboard

El problema Checkerboard [23] simula un tablero con $m \times n$ casillas, donde una función coste $c(i, j)$ devuelve el coste asociado a acceder a la casilla $(i, j)$ (siendo $i$ la fila y $j$ la columna). La meta de este código es encontrar el camino más corto (un camino es la suma de los costes de las casillas visitadas) para llegar desde la primera a la última fila, asumiendo que sólo se puede mover a los vecinos, diagonalmente o hacia arriba y siempre avanzando a la fila siguiente. Para calcular la solución, definimos en la ecuación 3.5 la función $q(i, j)$ como el coste mínimo para ir a la casilla $(i, j)$ desde la primera casilla. La función $c(i, j)$ es una función que devuelve el coste de

acceder a la casilla $(i, j)$ que puede variar según las características del problema. Para los experimentos de este capítulo la función coste seleccionada tiene aproximadamente 5000 operaciones enteras.

$$q(i,j) = \begin{cases} \infty & j < 0 \, o \, j > n-1 \\ c(i,j); & i = 0 \\ f(i,j); & \end{cases} \tag{3.5}$$

donde la función $f(i, j)$ es calculada como:

$$f(i,j) = min(q(i-1,j-1), q(i-1,j), q(i-1,j+1)) + c(i,j) \tag{3.6}$$

De hecho, identificaremos a cada tarea con la computación $q(i, j)$. En el código secuencial (figura 3.45) del algoritmo observamos como cada casilla necesita tres elementos de la fila anterior; lo que implica que de una casilla dependen tres vecinos: los que están a distancia (1,-1), (1,0) y (1,1).

A partir del análisis del código secuencial y de la ecuación que define el código del Checkerboard, podemos escribir el fichero de definición descrito en la figura 3.44 para implementar el problema con la plantilla. El método `executeTask` queda definido como muestra la figura 3.46, utilizando la misma función auxiliar, `Min`, definida en el código secuencia de la figura 3.45. Mientras que el método `dataInit` se muestra en la figura 3.47.

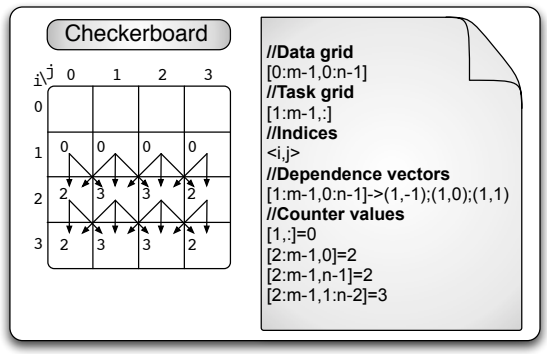



Figura 3.44: Fichero de definición del problema Checkerboard y patrón de dependencias.

```
int Min(int i, int j){
  int min_result=0;
  if ((i>0) && (j>0))
    min_result = q[i-1][j-1];
  if (i>0)
    min_result = min (min_result, q[i-1][j]);
  if ((i>0) && (j<n-1))
    min_result = min(min_result, q[i-1][j+1]);
  return min_result;
}
void main(){
  for (int i = 1; i<m; i++){
    for (int j=0; j<n; j++){
      q[i][j] = Min(i, j) + c(i, j);
    }
  }
}
```

Figura 3.45: Código secuencial del problema Checkerboard, donde f es una función aritmética.

```
int Min(int i, int j);

void Operation::executeTask() {
  int i  = GetFirst();
  int j  = GetSecond();
  q[i][j] = Min(i,j)+c(i,j);
}
```

Figura 3.46: Método `executeTask()` del problema Checkerboard.

```
int dataInit (int argc, char **argv)
{
 //input arguments managment
 q = malloc(size);
 for (int i=0; i<m; i++)   q[i][0] = 1000000000;
 for (int j=0; j<n; j++)   q[1][j] = f(1,j);
 return 0;
}
```

Figura 3.47: Función `dataInit()` del problema Checkerboard.

#### 3.3.3.3. Problema Financial

El problema Financial asume que dadas $m$ funciones $f_1, f_2, ..., f_m$ (cada una representa una función asociada a los depósitos realizados en cada banco $i$) y un entero positivo $n$ (el presupuesto disponible total), queremos maximizar la función $f_1(x_1)+f_2(x_2)+...+f_m(x_m)$ con la restricción $x_1+x_2+x_3+...+x_m = n$, donde $f_i(0) = 0(i = 1, .., m)$. Así que el objetivo es maximizar el interés financiero de depositar $n$ euros en $m$ bancos. Los valores $x_i$ representan la cantidad de esos $n$ euros a depositar en el banco i. Ahora, para solucionar el problema definimos una matriz $I$, $m \times n$ donde se almacenan los valores parciales. El valor $I(i, j)$ se define según la ecuación 3.7, obteniendo la solución para el problema en $I(m-1, n-1)$. Cada función $f_i$ también llamada función de interés del banco i, ejecuta alrededor de unas 5000 operaciones enteras para los experimentos de este capítulo.

$$I(i,j) = \begin{cases} f_1(j) & if\ i = 1 \\ \underset{0 \le t \le j}{max}\{I(i-1, j-t) + f_i(t)\} & eoc \end{cases} \tag{3.7}$$

Aquí, identificaremos una tarea como la computación de una casilla de la matriz $I(i, j)$. Analizando el código secuencial del problema (figura 3.48) vemos que de cada celda dependen los valores de todas las celdas desde la columna actual hasta el último elemento de la fila siguiente. Por tanto el vector de dependencias se indicaría mediante $(1, 0 : n-j-1)$. Esto significa que la carga computacional para cada celda depende de la columna de la celda que estamos calculando. Por tanto, el grano de tarea es variable. Las tareas correspondientes a las primeras columnas de la matriz tendrán un grano de tarea muy fino (la primera tarea tendrá aproximadamente 3 operaciones flotantes), mientras que las tareas que estén al final de cada fila tendrán una carga computacional muy grande ($3 * n$ operaciones flotantes). De esta manera una tarea del problema financial tendrán en media $(3 + 3 * n)/2$ operaciones flotantes más las operaciones que tenga la función de interés.

Con esta información construimos el fichero de definición que se presenta en la figura 3.49. En éste indicamos que las tareas de la región $[1 : m - 2, 1 : n - 1]$ tienen como dependencias el vector $(1, 0 : n - j - 1)$. Por ejemplo, en la figura 3.49 para la celda (2,1), las tarea dependientes son aquellas en la siguiente fila y en las columnas dentro del rango $1 + (0 : n - 1 - 1)$. así que las celdas dependientes son (3,1), (3,2) y (3,3) si n = 4.

```
1 double Max(int i, int j){
2   double max_result = I[i-1][j] + f(i,0);
3   for (int t=1; t<=j; t++){
4     unsigned int parcial= I[i-1][j-t]+f(i,t);
5     max_result = max (max_result, parcial);
6   }
7  return max_result;
8 }
9 void main(){
10 for (int i=1; i<m; i++){
11   for (int j=1; j<n; j++){
12    I[i][j] = Max(i,j);
13   }
14 }
```

Figura 3.48: Código secuencial del problema Financial donde `f` es la función de interés.

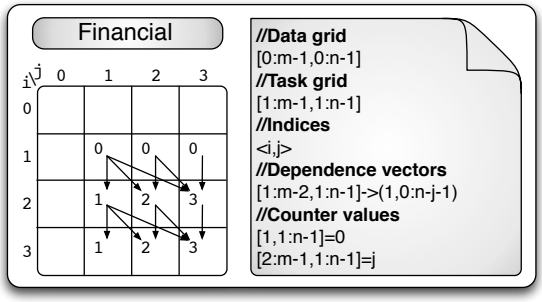



Figura 3.49: Fichero de definición del problema Financial y patrón de dependencias.

El método `executeTask` necesario para implementar el problema utilizando la plantilla está definido en la figura 3.50 y el método `dataInit` está presentado en la figura 3.51. Como dijimos anteriormente, podemos inferir la matriz de contadores del patrón de dependencias. La forma de inferir los contadores del patrón de dependencias, es realizar un recorrido en paralelo por todos los elementos de la matriz llamando al método `GetCounter` por defecto que llama a su vez a `GetDependency` tal y como se realiza en el método `launch`. Por cada una de las dependencias encontradas para cada elemento de la matriz, se incrementa en uno su contador. El problema es que cuantas más dependencias haya por tarea, más pesada será la iniacialización de los contadores. En este caso, al tener cada tarea $m - j$ dependencias, que consideramos un número alto, la inicialización automática es costosa. Por eso, en este problema indicar el valor de

inicialización de los contadores es muy importante y mucho más eficiente. En concreto, con una matriz de $300 \times 300$ el no indicar la inicialización de los contadores en el fichero de definición, implica que la inicialización sea 70 veces más lenta. Sin embargo, este es un caso extremo: para el problema clásico wavefront 2D (para el cual la inicialización automática implica el incremento de dos contadores por tarea como mucho), la inicialización de la matriz de contadores de $1000 \times 1000$ es solo 5 veces más lenta si no especificamos los valores iniciales de la sección 5 del fichero de definición.

```
1 double Max(int i, int j);
2
3 void Operation::executeTask() {
4   int i  = GetFirst();
5   int j  = GetSecond;
6   I[i][j] = Max(i,j);
7 }
```

Figura 3.50: Método `executeTask()` del problema Financial.

```
1 int dataInit (int argc, char **argv)
2 {
3   //Procesado de argumentos
4    I = malloc(size);
5    for (int i=1; i<m; i++)    I[i][0] = 0;
6    for (int j=1; j<n; j++)    I[1][j] = f(1,j);
7 }
```

Figura 3.51: Función `dataInit()` del problema Financial.

#### 3.3.3.4. Algoritmo de Floyd

El algoritmo de Floyd [64] utiliza programación dinámica para resolver el problema de encontrar el camino más corto en un grafo denso. El método hace un uso eficiente de una matriz de adyacencia $D(i, j)$ para resolver el problema. Los nuevos valores de la matriz son calculados siguiendo la ecuación 3.8. El código secuencial del algoritmo de Floyd está implementado en la figura 3.52

$$D_k(i,j) = \min_{k \geq 1}\{D_{k-1}(i,j), D_{k-1}(i,k) + D_{k-1}(k,j)\} \tag{3.8}$$

```
for(int k=1; k<m; k++){
  for (int i=0; i<m; i++){
    for (int j=1; j<m; j++){
      D[i][j] = min(D[i][j], D[i][k]+D[k][j]);
    }
  }
}
```

Figura 3.52: Código secuencial para el algoritmo de Floyd.

Analizando el código secuencial del algoritmo de Floyd vemos que tiene un triple bucle anidado, el cual $D(i,j)$ es sobrescrito por el bucle interno. Para la aproximación wavefront hemos eliminado la dimensión $j$ del espacio de tareas, así que cada tarea tiene que computar $D(i,:)$ filas en cada iteración `k` e `i`. Por este motivo, los índices de la región de tareas han sido definidos como `(k,i)`. En el fichero de definición para el algoritmo de Floyd se muestra en la figura 3.53. Destacamos la definición de los índices `<k,i>` y la subregión `[0:m-2,k+1]` identifica las celdas de la super diagonal: aquellas celdas que verifican que $i == k+1$ para `k=0:m-2`. Por otro lado, la subregión `[0:m-2,!(k+1)]` representa todas las celdas que verifican `i!=k+1`.

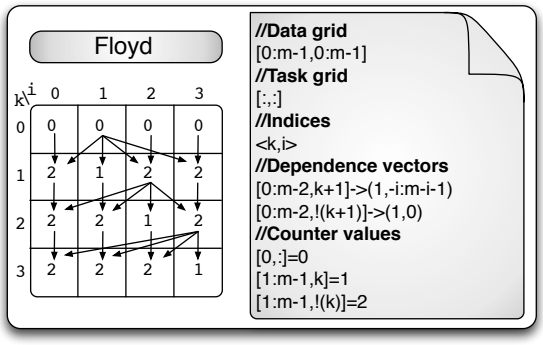



Figura 3.53: Fichero de definición para el algoritmo de Floyd y patrón de dependencias.

La figura 3.54 contiene el método `executeTask` para el algoritmo de Floyd. Como vemos cada tarea, realiza toda la iteración del bucle "j". En la figura 3.55 mostramos el método de inicialización del problema que consiste en la lectura de la matriz de distancia D.

```
1 void Operation::executeTask() {
2  int i  = GetFirst();
3  int k  = GetSecond();
4  for(int j=1;i<m;j++){
5   D[i][j] = Min(D[i][j], D[i][k]+D[k][j]);
6  }
7 }
```

Figura 3.54: Método `executeTask()` del algoritmo de Floyd.

```
1 int dataInit (int argc, char **argv)
2 {
3   //Procesado de argumentos
4   D = read_matrix();
5 }
```

Figura 3.55: Función `dataInit()` del algoritmo de Floyd.

#### 3.3.3.5. Decodificador H264: comparación con una implementación en pthreads

Por último, hemos implementado el decodificador H264. H264 o AVC (Advanced Video Coding) es un estándar para la compresión de vídeo.

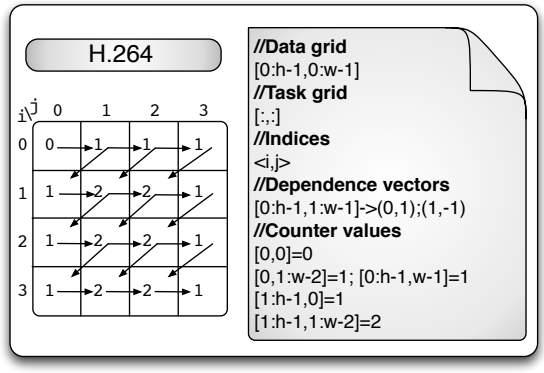



Figura 3.56: Fichero de definición y patrón de dependencias para el H264.

En H264 una secuencia de video tiene múltiples frames. Un frame está formado por varias imágenes, las cuales contienen una partición del frame que a su vez contiene va-

rios macrobloques (MB). Los macrobloques son bloques de 16 x 16 pixeles y son la unidad básica para codificar y decodificar. Los MB en un frame son procesados normalmente en el orden de escaneo, lo cual significa que empiezan en la esquina superior izquierda de un frame y se va moviendo hacía la derecha, de fila en fila. Sin embargo, procesando los MB siguiendo un patrón wavefront podemos satisfascer todas las dependencias y, al mismo tiempo, explotar paralelismo entre macrobloques.

```
1 while (1){
2   get_mb_from_taskq(fm, &mb_args);
3   if (mb_args.mb_xy < 0){
4 // Si recibimos un MB vacío de la cola de tareas: Se manda comunica el
      fin de lframe
5     barrier_execute(&fm->frame_barrier_end, fm->h->s.avctx->th_count-1);
6     break;
7   }
8 do {
9   mb_xy = mb_args.mb_xy;
10  curr_mb_blocks = mb_blocks + mb_xy;
11 // decodifica el actual mb
12  copy_mb_to_context(h_local, curr_mb_blocks);
13  hl_decode_mb(h_local);
14 // Actualiza la tabla de dependencias
15  update_mb_dependencies(fm, h_local, mb_xy, &mb_right_ready, &
        mb_down_left_ready);
16 // Introduce en la cola de tareas el siguiente mb preparado.
17  submit_ready_mb(fm, h_local, mb_xy, &mb_right_ready, &
        mb_down_left_ready, &mb_args);
18 } while (mb_right_ready || mb_down_left_ready);
```

Figura 3.57: Código wavefront en la versión Pthreads del H264.

En [55] se estudia la escalabilidad a nivel de Macrobloques del decodificador H264 para aplicaciones de alta definición (HD). En el trabajo se presenta un wavefront 2D para implementar una versión del H264, utilizando pthreads. En concreto, nos centramos en la función `tf_decode_mb_control_distributed` donde los macrobloques son procesados siguiendo un patrón wavefront. El código original de wavefront en [55] es un código basado en Pthreads, con un thread maestro y un pool de threads trabajadores que esperan trabajo en una cola de tareas. Cada thread esclavo realiza el trabajo descrito en la figura 3.57, mientras que el thread maestro es responsable del control de las operaciones de inicialización y de terminación. Las dependencias de cada macrobloque MB son expresadas por una tabla de dependencias. Cada vez que se satisfacen las dependencias para un macrobloque, se inserta una nueva tarea en la cola de tareas. La sincronización de los threads y los accesos al pool de tareas fueron implementadas utilizando semáforos. En la figura 3.57 presentamos el núcleo computacional de la versión del H264. Esta

versión está implementada en Pthreads. Se trata de un bucle que decodifica macrobloques mientras una cola gobal de macrobloques preparados no esté vacía. En la línea 2 se extrae el siguiente macrobloque a decodificar de la cola de macrobloques disponibles llamando a la función `get_mb_from_taskq`. Posteriormente, necesitamos copiar en el contexto el macrobloque actual para poder procesarlo (línea 12). En la línea 13 se realiza la decodificación del macrobloque. El siguiente paso es similar al que se realiza en la plantilla tras realizar el trabajo de cada tarea: actualizar los contadores de los bloques vecinos (utilizando la función `update_mb_dependencie` en línea 15 de la figura 3.57). Por último, se introducen los macrobloques vecinos que estén listos en la cola de tareas preparadas con la función `submit_ready_mb` en la línea 17. Con esta implementación como punto de partida hemos re-codificado este procedimiento del H264 aprovechando, nuestra plantilla para mejorar la eficiencia.

El fichero de definición presentando en la figura 3.56 expresa el patrón de dependencias descrito en [55] y [10]. Se define una matriz de datos de $h \times w$ macrobloques, MB. Y una tarea será la encargada de controlar cada macrobloque. Las dependencias de tareas son descritas con tan solo los vectores (0,1) y (1,-1). Remarcar que aunque las dependencias (1,0) y (1,1) también existen, esos vectores no son realmente necesarios, ya que están implícitos en la secuencia (0,1) y (1,-1). Sin embargo, en nuestra implementación, además del ya discutido fichero de definición, sólo necesitamos especificar el cuerpo de la tarea mostrado en la figura 3.58. Esto simplifica en gran medida la tarea de portar el código a TBB gracias a nuestra plantilla, eludiendo las responsabilidades de controlar la creación de threads o tareas, la gestión de la cola de tareas y la sincronización entre las mismas, por mencionar algunas. El trabajo que cada tarea realiza (figura 3.58) utiliza una zona privada de trabajo (`local_WS`) para cada thread. Esto significa, que, aunque puede haber muchas tareas en TBB, esas tareas serán llevadas a cabo por los threads que estén vivos; así que el número máximo de tareas ejecutándose simultáneamente es igual al número de threads. Usualmente, este número de thread es mucho más pequeño (normalmente en TBB es igual al número de cores, para evitar el overhead de oversubscription).

```
1 void Operation::executeTask() {
2   H264mb * curr_mb_blocks;
3   local_WS = get_Local_Workspace();
4   curr_MB = Get_curr_MB(GetFirst(),GetSecond());
5 // Decodifica el mb actual
6   copy_mb_to_WS(local_WS, curr_MB);
7   hl_decode_mb(local_WS);
8 }
```

Figura 3.58: Método `executeTask()` para el problema H264.

Utilizar un espacio de trabajo por thread en lugar de por tarea es mucho más eficiente, de este modo, la función `Get_Local_WorkSpace` (línea 3 en la figura 3.58) identifica cuál es el thread donde la tarea se está ejecutando y el correspondiente espacio local de trabajo. Entonces, utiliza la información del macrobloque actual identificado por los métodos de la plantilla `getFirst()` y `getSecond()` para acceder a él y proceder a su decodificación.

## 3.3.4. Resultados experimentales

A continuación presentamos dos conjuntos de experimentos. El objetivo del primero de ellos es validar la eficiencia de nuestra plantilla, mientras que el segundo pretende analizar la programabilidad. Hemos utilizado como benchmarks el código básico 2D, Checkerboard, el problema Financial, el algoritmo de Floyd y el decodificador H264 que anteriormente describimos. No incluimos en estos experimentos el problema smith-watterman debido a que su grano de tarea es muy pequeño (apenas 10 FLOP por tarea) y como ya estudiamos en el primer apartado del capítulo, necesitamos un grano de tarea mayor para poder obtener rendimiento. Sin embargo, en el próximo capítulo abordaremos unas optimizaciones que resuelven este problema.

### 3.3.4.1. Overhead de la plantilla

Para estos experimentos, hemos utilizado una máquina multicore con 8 cores, donde cada core es un Intel Xeon® CPU X5355 a 2,66 GHz, con un sistema operativo SUSE LINUX 10.1. Los códigos fueron compilados con icc 12 -O3. Para obtener los resultados ejecutamos cada prueba 10 veces y obtuvimos el resultado medio de los experimentos. Elegimos como medida el speedup y lo calculamos respecto al tiempo secuencial de cada código.

En el primer experimento analizamos en detalle el comportamiento de la plantilla para nuestro problema básico 2D. En este experimento el objetivo era estudiar la escalabilidad de nuestra plantilla para diferente granuralidad de tarea.

En la figura 3.59 podemos observar el speedup para dos implementaciones del código 2D en tres granuralidades: i) TBB-Fine, TBB-Medium y TBB-Coarse representan el speedup para la implementación manual (ver código de la figura 3.20); ii) Template-Fine, Template-Medium y Template-Coarse representan el speedup para la implementación del problema utilizando nuestra plantilla (los detalles de la implementación están mostrados en la figura 3.29). En los experimentos, la granuralidad de una tarea es controlada por el valor de `gs` en la función `foo` (línea 6 de la figura 3.29). Estos parámetros actualizan el número de operaciones en punto flotantes que realizará cada tarea. El gran

fino (Fine) de tarea representan 200 FLOP, el grano medio (Medium) representa 2000 FLOP y el grano grueso (Coarse) 20.000 FLOP. En todos los casos, la matriz de datos ha sido de $10.000 \times 10.000$ celdas. En los resultados de la figura 3.59, podemos ver claramente que la granuralidad gruesa es en la que mejor escalan los códigos (en ambos casos, versión manual y con la plantilla), siendo las diferencias de rendimiento entre ambas implementaciones pequeña. Probando con otros tamaños de matrices observamos que los resultados son similares.

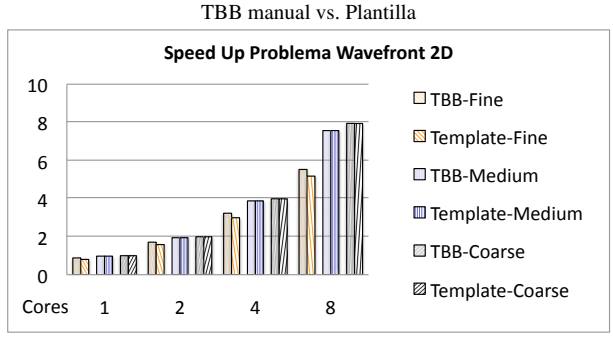



Figura 3.59: Speedup para el problema básico de wavefront 2D para diferente granuralidad de tarea: Fine (fina)=200, Medium (media)=2,000 y Coarse (gruesa)=20,000 FLOP). Comparamos el speedup de la implementación manual con el conseguido con la plantilla para 1, 2, 4, y 8 cores.

La figura 3.60 representa el overhead que introduce la plantilla para la implementación del problema wavefront 2D en sus diferentes granuralidades. El overhead se calcula como la ratio entre la diferencia de tiempos de la plantilla y la versión manual respecto al tiempo de la versión manual. En otras palabras, el incremento de tiempo sobre la versión manual. De la figura se extrae que el overhead no es significativo para el grano grueso, y que incluso para la granuralidad media el overhead es menor del 2 %. Sin embargo, en el grano fino, el overhead debido a la plantilla incrementa el tiempo de ejecución desde un 6 % a un 9 %. En la figura 3.61 se muestra la contribución de overhead debido a las distintas etapas de la plantilla: la etapa de inicialización (`wavefront_init` en la línea 5 de la figura 3.29) y la etapa de computación `wavefront_run` en la línea 6 de la figura 3.29). La etapa de computación representa la principal fuente de overhead en cualquier grano de tarea hasta 4 cores. Sin embargo, a partir de 4 cores la etapa de inicialización es más costosa ya que como vimos, para cada elemento hay que comprobar si su contador es 0 para añadir la tarea preparada, acción que no se realiza en la versión manual.

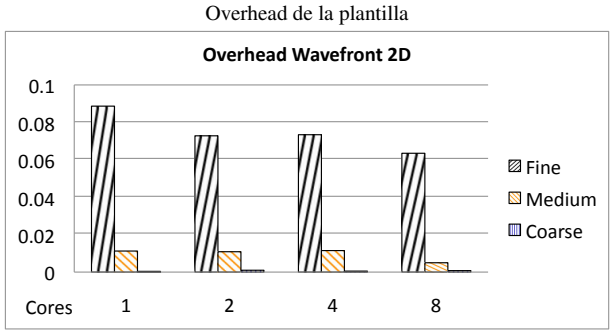
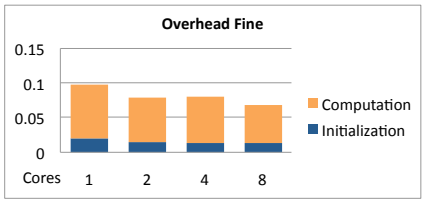
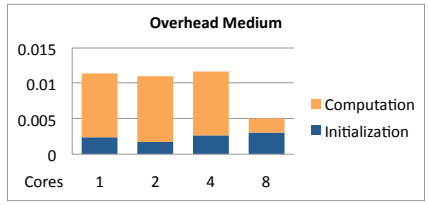


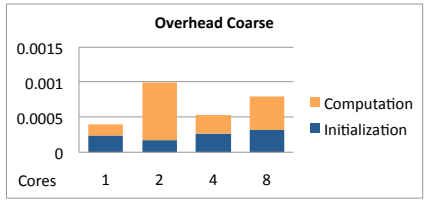

Figura 3.60: Overhead de la plantilla para el problema wavefront básico 2D y diferentes granuralidades.




(a) Fina




(b) Media




(c) Gruesa

Figura 3.61: Contribuciones al overhead debido a la plantilla: etapa de inicialización y etapa de cómputo. El eje de abscisas representa el número de cores.

En el próximo experimento, nos centramos en los problemas reales Checkerboard, Financial y Floyd. Hemos seleccionado distintos tamaños de matriz para cada problema. Para Checkerboard escogimos tamaño de $1500 \times 1500$ celdas, para el problema Financial una matriz de $300 \times 300$ y para el algoritmo de Floyd $5000 \times 5000$.

En la figura 3.62 mostramos el speedup para dos implementaciones de cada código. TBB X (donde X es el nombre de cada problema) representa el speedup para la implementación manual de TBB y Template X representa el speedup para la implementación basada en la plantilla. En estos resultados vemos nuevamente que las diferencias entre la plantilla y la implementación manual son pequeñas; aunque la versión manual de TBB tiende a escalar mejor. Los resultados son similares, aunque cambiemos los tamaños de las matrices.

Versión manual de TBB vs. La plantilla.

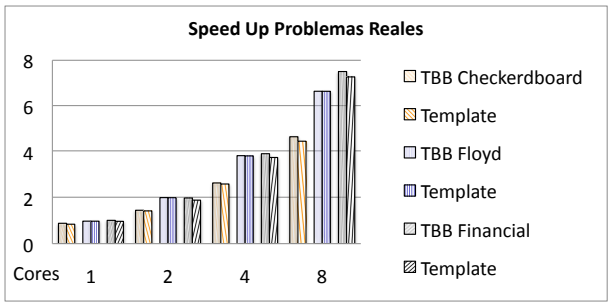



Figura 3.62: Speedup para los códigos Checkerboard, Financial y Floyd.

Estos tres problemas exhiben una granuralidad fina o media, y con desbalanceo de carga entre las tareas de los problemas Financial y H264. Como ya ocurriera con el problema básico de wavefront 2D, para estos problemas también comprobamos que a menor tamaño de grano, peor rendimiento. En concreto, Checkerboard se corresponde con el grano más fino de tarea, lo cual explica su baja escalabilidad. Se estudia a continuación el motivo por el que obtenemos el peor rendimiento en los problemas de grano fino.

La figura 3.63 representa el overhead debido a la utilización de la plantilla en los problemas Checkerboard, Financial y algoritmo de Floyd. El overhead se ha calculado como la ratio entre la diferencia de tiempos de la implementación de los problemas utilizando la plantilla y la versión manual. Podemos ver en la figura que el overhead de la plantilla es pequeño situándose entorno al 5 % en el código del problema Financial y menor del 0,5 % en el algoritmo de Floyd. Para entrar en detalles, la figura 3.64 ilustra la contribución debida a la inicialización y cómputo en esos códigos. El overhead de la inicialización tiene un impacto pequeño especialmente en Checkerboard y en el código Financial. El tamaño de la matriz de datos explica estos resultados, ya que son más pequeños (y por tanto tiene menos contadores) que la utilizada en el algoritmo de Floyd. Resumiendo, estos experimentos han mostrado que la implementación de nuestra plantilla es bastante eficiente para una variedad de códigos wavefront. El coste de abstracción

por usar nuestra plantilla supone un incremento de los tiempos de ejecución del 1 % en problemas de grano grueso e inferior al 8 % en problemas de grano fino. Creemos que el coste es pequeño comparado con la contribución que hace la plantilla para mejorar la productividad del programador, aspecto que discutimos en la próxima sección.

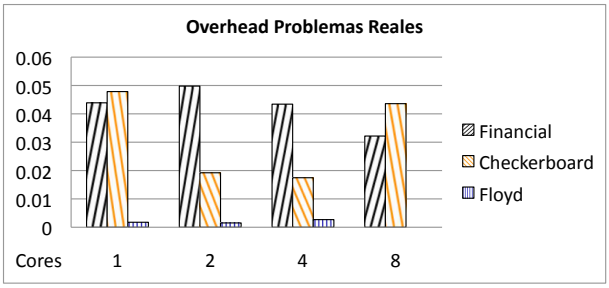
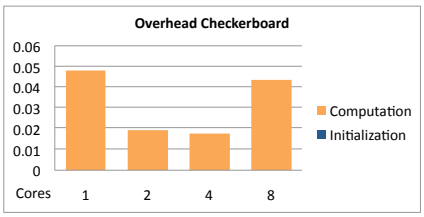
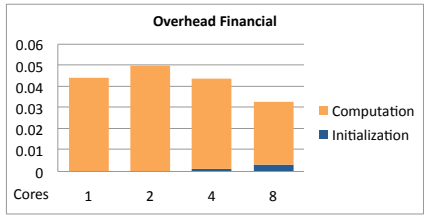


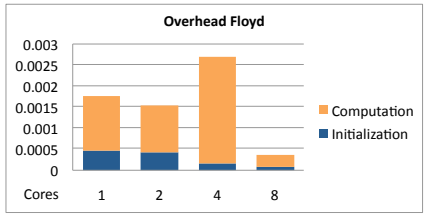

Figura 3.63: Overhead de la plantilla para los códigos Checkerboard, Financial y Floyd.




(a) Checkerboard



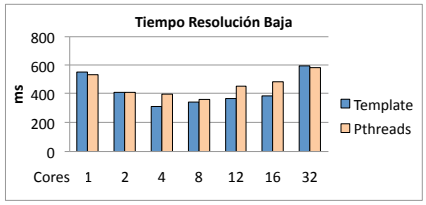

(b) Financial



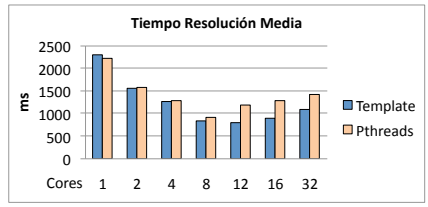

(c) Floyd

Figura 3.64: Contribuciones al overhead debido a la plantilla: etapa de inicialización y etapa de cómputo para los problemas reales. El eje de abscisas representa el número de cores.

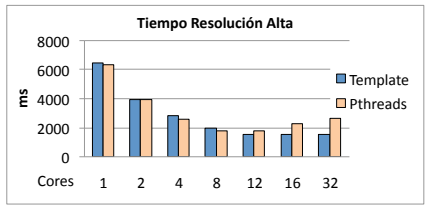
Para evaluar nuestra plantilla, con un problema más grande y complejo, presentamos un nuevo experimento donde comparamos dos implementaciones del benchmark de decodificación de video H264: la implementación original basada en Pthreads [55] respecto la versión basada en nuestra plantilla. Ambas implementaciones del H264 fueron ejecutadas en una plataforma: con 4 procesadores de 8 cores Intel Xeon ® CPU X7550 a 2 GHz (32 cores) cuyo sistema operativo es el SUSE 11.1. Los códigos fueron compilados con g++ 4.1. Ejecutamos cada versión 10 veces y obtuvimos la media del tiempo de ejecución. En particular, medimos el tiempo de ejecución de la decodificación de los macrobloques (MB). Ejecutamos cada versión para distinta resolución de frame del mismo vídeo. La meta era evaluar el impacto en diferentes tamaños de entrada del problema en la escalabilidad para cada versión. En la figura 3.65 vemos el tiempo (en ms). Para las dos versiones, utilizamos como datos de entrada tres resoluciones: baja resolución ($352 \times 288$ píxeles = 396 MB), resolución media ($704 \times 576$ pixeles = 1584 MBs) y alta resolución ($1280 \times 720$ pixeles = 3600 MB).




(a) Baja resolución: $352 \times 288$




(b) Media resolución: $704 \times 576$




(c) Alta resolución: $1280 \times 720$

Figura 3.65: Tiempo en ms para las versiones de Pthreads y de la plantilla TBB para el código H264 y diferentes resoluciones de frame.

De los resultados, podemos extraer que el overhead introducido por nuestra plantilla es despreciable. Además, nuestra plantilla TBB tiene una mejor escalabilidad para todas las resoluciones de vídeo. Aunque para pocos cores (1, 2 cores) ambas implementaciones tienen resultados similares. Sin embargo, para un mayor número de cores (12, 16 y 32 cores), el rendimiento de la implementación de Pthreads tiende a degradarse más rápido

que nuestra implementación utilizando la plantilla especialmente para las resoluciones media y alta. En esos casos, la contención de la cola de tareas de Pthreads es una de las razones que explica la degradación que sufre su implementación. Este problema no existe en nuestra implementación, gracias a la cola distribuida de tareas del modelo de TBB. De cualquier modo, la poca carga de trabajo explica la escalabilidad para la resolución baja y media.

#### 3.3.4.2. Programabilidad de la plantilla

En este apartado discutiremos los hallazgos logrados con nuestro segundo experimento: evaluar la programabilidad de la plantilla. En otras palabras, cómo de productivo es para un programador utilizar la plantilla para códigos wavefront. Es difícil medir la facilidad de programación. Para ello, hemos seguido una metodología propuesta en [33], donde los autores sugieren tres métricas cuantitativas para medir la facilidad de programación de un código. Estas métricas son: SLOC (*source lines of code*, líneas de código fuente), CC (*cyclomatic complexity*, complejidad ciclomática) y PE (*programming effort*, esfuerzo de programación). Cuando contamos los SLOC, no consideramos los comentarios ni las líneas en blanco. Esta métrica es, quizás, más dependiente del estilo de programación del programador que las otras dos medidas. En general, podemos asumir que valores altos de esas métricas implican una posibilidad mayor de error y una dificultad de mantenimiento más alta. Con respecto al CC, los autores en [54] definen esta métrica como el número de predicados o puntos de decisión de un programa más uno. El número ciclomático se corresponde con el número máximo de caminos independientes que existen en un programa estructurado. Valores altos de este parámetro significan un código más complejo. Por último, el parámetro PE, está definido en [37] como una función del número de operadores únicos, total de operandos y total de operadores encontrados en el código. Los operandos se corresponden a las constantes y a los identificadores, mientras que los símbolos o combinaciones de símbolos que afectan a los valores de los operandos constituyen los operadores. Esta métrica puede ser representativa del esfuerzo de programación requerido para implementar el algoritmo. Así que un valor alto de PE significa que es más difícil para un programador codificar el algoritmo.

La figura 3.66 muestra los resultados de las métricas de programabilidad para los cuatro problemas reales de wavefront: el problema Checkerdboard, el problema Financial, el algoritmo de Floyd y el decodificador H264. Comparamos las métricas de dos implementaciones de cada código: la versión manual en TBB (ver Fig. 3.20) para el problema Checkerboard, el problema Financial y el algoritmo de Floyd, y la original de Pthreads para H264, con la versión de cada código utilizando nuestra plantilla. Los valores de SLOC, CC y PE están normalizados respecto a los correspondientes valores de las

implemetaciones manuales. Para todos los códigos, las métricas correspondientes a las versiones utilizando la plantilla tienen siempre valores más pequeños. Por ejemplo, los SLOC, los CC y PE para los códigos de la plantilla está sobre el 50 % de las versiones de las implementaciones manuales de TBB; para H264, la métrica PE para nuestra plantilla es un 25 % menor que la implementación Pthreads. En cualquier caso, esos resultados indican que existe ganancia en cuanto a facilidad de programación cuando utilizamos la plantilla.

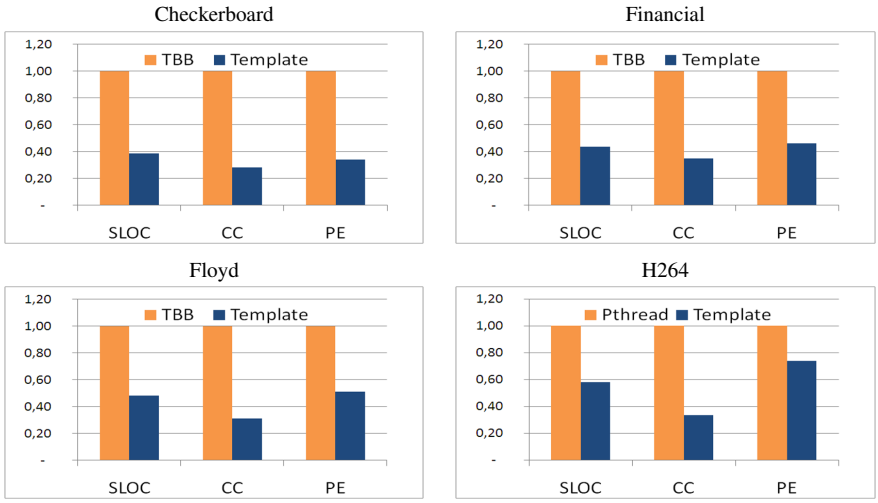



Figura 3.66: Métricas de programabilidad para los códigos Checkerboard, Financial, Floyd y H264, comparando las versiones manual y utilizando la plantilla.

Resumiendo, la evaluación experimental desarrollada en esta sección ha mostrado que nuestra plantilla basada en TBB para problemas de tipo wavefront constituye una herramienta productiva con un bajo coste de overhead, cuando codificamos aplicaciones complejas.

## 3.4. Trabajos relacionados

Existen diversos trabajos que han tratado la paralelización de problemas de tipo wavefront. En [47], los autores proponen el concepto de la programación basada en regiones y describen los beneficios de expresar a alto nivel computación sobre arrays en el contexto del lenguaje de paralelización ZPL, siendo wavefront un problema particular

de este patrón. Aunque la definición de las regiones en este trabajo es similar a la que hemos propuesto en la tesis, la nuestra se diferencia en el objetivo. Los autores en [47] utilizan la información proporcionada por las regiones para identificar el patrón en una arquitectura de memoria distribuida. Sin embargo, en nuestro trabajo queremos reducir el esfuerzo del programador a la hora de expresar la información de dependencias del problema.

Otros autores han abordado el problema wavefront tanto en sistemas de memoria distribuida como en sistemas heterogéneos. En todos estos trabajos los autores han realizado implementaciones paralelas del problema wavefront en arquitecturas heterogéneas, como aceleradores de tipo SIMD[76], o la arquitectura *Cell/BE* [1]. Estos autores han centrando sus esfuerzos en las optimizaciones de vectorización y en estudiar la distribución apropiada del trabajo en cada arquitectura, forzando al programador a trabajar a un bajo nivel de detalle. Claramente, esto es contrario a nuestra intención de programar a un alto nivel liberando al usuario de conocer los detalles de bajo nivel de la arquitectura.

Precisamente, para liberar al usuario de trabajar en ese nivel bajo, existen también trabajos que han estudiado la caracterización de patrones de programación paralela o "skeletons" [28], siendo wavefront uno de esos patrones [4]. En esta línea, en el trabajo [85] los autores proponen una abstracción wavefront para un cluster multicore. Nos diferenciamos de este trabajo en que los problemas de wavefront para este patrón necesitan un grano grueso de computación (una celda necesita alrededor de 117 segundos en una CPU de 1Ghz). Ellos utilizan threads y confían en el planificador del sistema operativo. Por el contrario, nuestro estudio está basado en un grano de tarea más fino y en problemas regulares de wavefront, así como en una programación basada en planificadores a nivel de tarea.

Una característica diferenciadora de nuestras aproximaciones con respecto al resto de trabajos previos es que proponemos una plantilla en el contexto de una librería moderna basada en el modelo de programación de tareas. Es decir, hemos identificado las características de bajo nivel necesarias para programar eficientemente problemas de tipo wavefront para ofrecer un interfaz de alto nivel que evita al usuario conocer dichas características de la arquitectura al tiempo que codifica una aplicación wavefront optimizada.

## 3.5. Conclusiones

A través de nuestro estudio de diferentes implementaciones de problemas wavefront mediante varios paradigmas basados en tareas, hemos encontrado que TBB posee una serie de características que permiten implementaciones más eficientes. Estas característi-

cas son las operaciones atómicas y el reciclado de tareas priorizando el reciclado en las tareas de la misma fila, explotando la localidad de datos y demostrando una gran mejora de rendimiento. En nuestra opinión estas características, deberían estar disponibles a nivel de usuario (como en TBB) para permitir optimizaciones a alto nivel, guiando al programador, en los otros lenguajes o extensiones de lenguajes que implementan el modelo basado en tareas. En cualquier caso, estas optimizaciones podrían estar encapsuladas en una plantilla de alto nivel que facilitase la programación de problemas tipo wavefront como las plantillas que posee TBB (`paralle_do o pipeline`), que encapsulan las mencionadas optimizaciones y evitan el manejo de las dependencias de los problemas wavefront (captura atómica, contadores ) y la gestión de tareas (`spawns`, reciclado, etc.).

El patrón de programación paralela wavefront es un paradigma que encaja perfectamente con la librería TBB basada en tareas. Sin embargo, TBB no ofrece una plantilla para este patrón . Por ello, en este capítulo proponemos una plantilla TBB para wavefront que permite al programador no tener que controlar la gestión a nivel de tareas como: i) la sincronización a través de la captura atómica; ii) la creación y el lanzamiento de nuevas tareas cuando una tarea está preparada para ser ejecutada y iii) la priorización de tareas que puedan explotar la localidad espacial.

Para utilizar nuestra plantilla, el programador sólo necesita escribir un fichero de definición con la descripción del patrón de dependencias y un fichero de cabecera indicando el trabajo que cada tarea debe realizar. Utilizando varios benchmarks hemos encontrado que el coste debido a la plantilla sólo supone alrededor del 5 % de overhead cuando lo comparamos con implementaciones manuales en TBB del mismo código. Por otro lado, en el caso del código H264, la versión de la plantilla mejora el rendimiento de la versión original implementada en Pthreads debido a las ventajas del planificador work-stealing de la librería de TBB.

Adicionalmente, hemos evaluado la programabilidad de nuestra plantilla en varios código complejos. Utilizando tres métricas cuantitativas que caracterizan la facilidad de programación. Encontramos que los códigos basados en la plantilla reducen el esfuerzo del programador entre un 25 % y un 50 %, cuando lo comparamos con la versión manual. Por todo ello, para los códigos evaluados estimamos que nuestra plantilla es una herramienta productiva con un coste pequeño en términos de overhead.

# 4 Optimizaciones de la plantilla wavefront

El objetivo fundamental de este capítulo es dar mayor funcionalidad y eficiencia a la plantilla para implementar problemas de tipo wavefront descrita en el capítulo anterior. Resumiendo, hasta ahora hemos visto que el modelo de programación basado en tareas se adapta mejor que el modelo de programación basado en threads para implementar los problemas de tipo wavefront. También que TBB es el modelo de programación basado en tareas que mayor rendimiento ha proporcionado en nuestros experimentos. Sin embargo, aunque TBB dispone de plantillas productivas y eficientes para implementar "parallel for" o "pipeline", no ofrece una plantilla similar que permita implementar cómodamente códigos de tipo wavefront. Para cubrir esa necesidad, en 3.3 presentamos una plantilla implementada en TBB para asistir al programador en la paralelización de los problemas de tipo wavefront. En los resultados experimentales, concluimos que nuestra plantilla reduce hasta un 50 % la complejidad para programar este tipo de códigos. En otros experimentos se comprueba que el uso de la plantilla apenas penaliza el tiempo de ejecución con respecto a la implementación manual de los mismo códigos.

No obstante, tanto si usamos la nueva plantilla como si implementamos a mano el wavefront, la eficiencia paralela no es aceptable cuando el grano de tarea es fino. En la sección 3.1 se explica cómo los problemas cuyas tareas realizan menos de 200 FLOPs no tienen una escalabilidad óptima. Los motivos que perjudican al rendimiento cuando el grano de tarea es fino son: i) el número de tareas, ya que el mayor overhead se concentra en el uso de la función `spawn` (una vez por tarea) y ii) la inicialización de los contadores ya que en algunos casos representa un porcentaje alto del tiempo total de ejecución. En 3.1 propusimos algunas optimizaciones que además incorporamos a nuestra plantilla: utilizar el mecanismo de reciclado de tareas de TBB y priorizar el reciclado en las tareas que aprovechen la localidad espacial. Sin embargo, recurrir a estrategias

de *tiling* y mejorar la fase de inicialización de los contadores son optimizaciones de la plantilla que se abordan en este capítulo. Además, caracterizaremos el rendimiento de los distintos tamaños de bloques posibles para hacer tiling y explicaremos como utilizar una versión mejorada de la plantilla que contempla estas dos nuevas optimizaciones.

## 4.1. Motivación

Antes de entrar en materia, presentamos en esta sección los resultados experimentales que motivan las optimizaciones que proponemos en el resto del capítulo. Hasta ahora, el coste computacional de cada celda de la matriz que recorremos en frente de onda ha sido como poco de 200 FLOPs. Pero, ¿qué pasaría si el grano fuera aún más fino?. Digamos, por ejemplo, de entre 5 y 10 FLOPs. Esta situación es también realista ya que por ejemplo, el código de Smith-Waterman presenta una granularidad de apenas 4 FLOPs y una operación entera. También en los problemas Checkerboard y Financial es posible encontrar una función de coste, $c$, o una función de interés, $f$, respectivamente, con granularidad inferior a 10 FLOPs. Si además, la arquitectura paralela dispone de mas cores, los problemas de escalabilidad en estas situaciones de grano muy fino se hacen muy patentes.

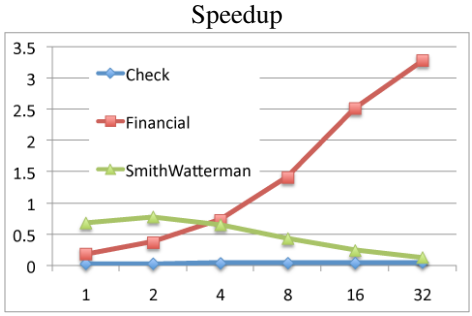



Figura 4.1: Speedup de los códigos Smith-Waterman, Financial y Checkerboard utilizando la plantilla.

Por tanto seleccionamos para este experimento una arquitectura de 32 cores y los códigos de Smith-Waterman, Checkerboard y Financial, los dos primeros con granularidad inferior a 10 FLOPs y Financial con una granularidad media de 5 KFLOPs. Las dimensiones de las matrices son de $40.000 \times 40.000$, $40.000 \times 40.000$ y $10.000 \times 10.000$, respectivamente. Con más detalle, la máquina multicore tiene cuatro procesadores Intel

Xeon® CPU X7550, a 2GHz y de 8 cores cada uno (total, 32 cores). El sistema operativo es SUSE LINUX Enterprise Server 11(X86_64). Todos los códigos fueron compilados con icc versión 12 con el flag de optimización -O3.

El speedup obtenido tras la ejecución de estos tres problemas se muestra en la figura 4.1, donde podemos ver como se comportan estos tres códigos de grano muy fino para un número de cores de 1 a 32. En Checkerboard, el speedup es siempre inferior a uno. En cuanto al problema Financial, observamos que sí escala algo, pero alcanza un speedup máximo inferior a 3.5 para 32 cores. Este problema, al realizar las operaciones y llamar a la función de interés j veces por tarea, tiene un mayor número de FLOPs y operaciones enteras (su grano de tarea es muy superior al del Checkerboard). Por último, Smith-Waterman se degrada conforme aumentamos el número de cores con un speedup inferior a uno. Estos problemas, además de ser de grano fino, son de intensidad aritmética baja, por lo que su rendimiento se ve perjudicado por la latencia en el acceso al bus de memoria.

Para mejorar los resultados obtenidos para grano fino, en este capítulo presentamos las optimizaciones necesarias para conseguir un mejor rendimiento en la implementación de problemas con poca carga computacional. En trabajos previos [27] encontramos que la inicialización de los contadores es una gran fuente de overhead. Esta inicialización es absolutamente necesaria para tener toda la información de las dependencias. Cada tarea tiene un contador, y en éste se guarda el número de tareas de las que depende. Cuando este contador toma el valor cero significa que la tarea asociada está preparada para ser ejecutada. Estos contadores deben ser inicializados con la información del fichero de definición, y son calculados en un bucle paralelo que recorre todos los elementos de la matriz. Si el fichero de definición especifica el valor de los contadores el proceso es más rápido ya que sólo hay que inicializar cada contador al valor especificado en el fichero de definición. Sin embargo la sección de definición de contadores es opcional y de hecho es recomendable no usarla ya que un usuario de la plantilla podría por error crear ficheros de definición incorrectos (si la especificación de contadores no se corresponde con la especificación de dependencias). Por tanto, no es sólo más cómodo, sino también deseable que el valor inicial de los contadores se infiera automáticamente a partir del patrón de dependencias especificado en el fichero de definición. Ahora, si el valor de los contadores hay que calcularlo, el coste de la inicialización puede ser bastante alto. En este caso, los contadores se inicializan a cero, y posteriormente cada contador se incrementa en uno por cada celda de la que depende. Este proceso puede ser más o menos lento dependiendo del número de dependencias que tenga cada elemento de la matriz. En algunos casos, encontramos que el tiempo de inicialización puede llegar a ser el 8 % del tiempo total de ejecución. Otra fuente de overhead que hemos encontrado, es que en algunos problemas (como Financial) cada tarea tiene muchas dependencias (en este caso $n - j - 1$ dependencias). Por tanto, cuando la tarea ha terminado de realizar el trabajo,

debe decrementar los contadores de esas $n - j - 1$ dependencias. El coste de gestionar esta actualización de las dependencias aumenta con el número de éstas, por lo que el rendimiento está directamente relacionado con su número de dependencias.

En definitiva, el objetivo del capítulo es optimizar la plantilla con las siguientes optimizaciones:

1. En lugar de inicializar todos los contadores antes de comenzar la ejecución, la inicialización se hará sólo para los contadores estrictamente necesarios. El resto de los contadores se irán inicializando en el transcurso de la ejecución wavefront.
2. Implementar una estrategia de tiling para aumentar el grano de tarea, mejorar la localidad espacial y reducir el número de tareas (lo cual a su vez, reduce el número de contadores y el overhead asociado)
3. Reducir el número de dependencias de forma que cuando una tarea termine no consuma mucho tiempo decrementando los contadores de las tareas dependientes.

Las tres siguientes secciones abordan cada uno de los tres puntos que acabamos de mencionar.

## 4.2. Inicialización de contadores

La inicialización de los contadores es una de las fuentes de overhead que encontramos en los experimentos realizados. Con la intención de minimizar el impacto de este problema, inicializaremos antes de la ejecución del código wavefront, sólo los contadores que son estrictamente necesarios para comenzar la computación. El resto de los contadores se inicializarán durante la ejecución del wavefront. Para ello, agruparemos un conjunto de contadores en lo que llamaremos macrobloques, como podemos ver en la figura 4.2, donde cada macrobloque agrupa a cuatro contadores.

Un macrobloque es una matriz de $mbs \times mbs$ contadores asociados a tareas adyacentes. La idea es la siguiente. Al igual que antes, el algoritmo para inicializar los contadores se ejecuta cuando el usuario realiza la llamada a `wavefront_init()`. La diferencia estriba en que en lugar de inicializar toda la matriz de contadores, inicializaremos sólo uno o varios macrobloques en los que, según los vectores de dependencias, al menos haya una celda que no depende de nadie. Completada esta primera inicialización, las tareas sin dependencias del macrobloque pueden despacharse, y éstas irán despertando a las demás. El resto de macrobloques de contadores se irán inicializando de forma solapada con la ejecución de tareas del wavefront. Para ello, hemos contemplado dos aproximaciones.

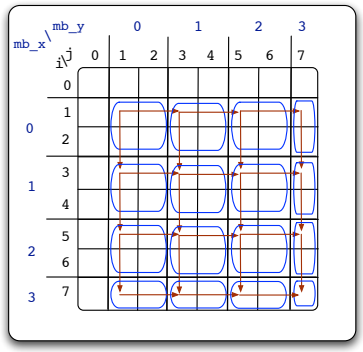



Figura 4.2: Macrobloques de contadores de inicialización.

Una posibilidad es que la primera tarea que tenga que decrementar un contador de una celda fuera de su macrobloque será responsable de, previamente, inicializar ese macrobloque. A nivel de implementación, eso implica una matriz con un contador atómico por cada macrobloque. Inicialmente el contador está a 1. La tarea que al decrementarlo lo haga igual a cero, es la encargada de inicializar todos los contadores del macrobloque. Además hace falta que las demás tareas que intenten acceder al mismo macrobloque esperen hasta que esté actualizado. Ya que esta alternativa presenta el coste adicional de la matriz con un elemento atómico por cada macrobloque, junto con el mecanismo de sincronización adicional, descartamos esta posibilidad.

La alternativa finalmente implementada consiste en que sólo una tarea del macrobloque inicialice los macrobloques vecinos antes de lanzar la ejecución del wavefront en ese macrobloque. De esta forma, cuando las tareas necesiten decrementar contadores de macrobloques vecinos, estos ya estarán inicializados. La elección de qué tarea será la elegida dentro del macrobloque para la tarea de inicialización se determina a partir de los vectores de dependencias. Estos marcan la dirección dominante del wavefront y con esta información se puede inferir cuál es la primera tarea a ejecutar en cada macrobloque y asignarle a ésta la misión de inicializar primero los macrobloques vecinos. Por ejemplo, en un problema 2D clásico con dependencias $(0, 1)$ y $(1, 0)$, se selecciona a la tarea que tenga coordenadas $(i, j)$ más pequeñas dentro del macrobloque.

Ilustramos a continuación el código generado automáticamente para el caso del fichero de dependencias correspondiente al problema Smith-Waterman, en el que la sección opcional de declaración de contadores se ha omitido intencionadamente. Recordemos que en este problema los vectores de dependencia son $(0, 1)$ y $(1, 0)$. Por tanto, y como vemos en la figura 4.2, las flechas saliendo desde la tarea de la esquina supe-

rior izquierda de cada macrobloque apuntan a los macrobloques adyacentes en los que hay que inicializar los contadores. En esta figura se usa `mb_x` y `mb_y` como índices de macrobloque, y vemos como por ejemplo, la tarea superior izquierda del macrobloque (1,1) inicializa los macrobloques (1,2) y (2,1). Insistimos en que esta inicialización se realiza antes de lanzar las tareas listas para despachar, por tanto antes de llamar a `executeTask`.

El pseudocódigo de la inicialización de macrobloques se puede ver en la figura 4.3: en la línea 1, la tarea llama a la función `am_I_the_first_task_of_mb` para saber si es la tarea seleccionada para inicializar macrobloques vecinos; en caso positivo, primero se computa cuales son las coordenadas del macrobloque que contiene a esa tarea, lo que permite calcular las coordenadas de los macrobloques vecinos que son inicializados en las líneas 3 y 4.

```
1 if (control->am_I_the_first_task_of_mb(i,j)){
2   int mb = control->getMyMacroBlock(i,j);
3   initializeCounterMB(mb.x, mb.y+1);
4   initializeCounterMB(mb.x+1, mb.y);
5 }
```

Figura 4.3: Pseudocódigo de la tarea de inicialización de macrobloques vecinos.

El pseudocódigo del método `initializeCounterMB(mb_x, mb_y)` se muestra en la figura 4.4. Recordando que los macrobloques son de $mbs \times mbs$, en el código vemos como dos bucles anidados recorren todas las coordenadas de los contadores del macrobloque. En el cuerpo del bucle se inicializa el contador de cada tarea asociada a esas coordenadas mediante una llamada a la función `GetCounter(taskID)`.

```
1 initializeCounterMB(mbx, mby){
2   int mbi = mbx*mbs;
3   int mbj = mby*mbs;
4   for (int i = mbi; i < mbi+mbs && i < m; i++){
5     for (int j = mbj; j < mbj+mbs && j < m; j++){
6        int taskID = coordinatesToID(i,j);
7        counter[taskID] = this->getCounter(taskID);
8     }
9   }
10 }
```

Figura 4.4: Pseudocódigo de `initializeCounterMB()` para la inicialización de los contadores de un macrobloque.

Por último describimos el método `GetCounter(ID)` que debe calcular el con-

tador inicial para la tarea `ID` (de cuantas tareas depende la tarea `ID`). Para ello necesitamos inferir automáticamente de cuantas tareas depende cada tarea del grid. Una posible implementación consiste en, dada cada región de dependencias (definido en la sección 4, `dependency vectors` del fichero de definición como se muestra en la sección 3.3.1), calcular que regiones resultan de aplicar los vectores de dependencias. Con esta información podemos luego inicializar el contador de cada tarea al número de regiones en las que la tarea está incluida.

Por ejemplo, en el problema Smith-Waterman la información de dependencias es `[1:m-1, 1:m-1] -> (0,1); (1,0)`. Aplicando la dependencia `(0,1)` a la región `[1:m-1, 1:m-1]` obtenemos la región `A=[1:m-1, 2:m]` y aplicando el vector `(1,0)` se obtiene la región `B = [2:m, 1:m-1]`. Ahora, las tareas que sólo pertenecen a la región `A` o (exclusivo) a la región `B` deben inicializar su contador a 1, mientras que si una tarea pertenece a las dos regiones simultáneamente, su contador se debe inicializar a 2. Note que dado que la región de tareas es `[1:m-1, 1:m-1]`, realmente las regiones `A` y `B` quedan en `[1:m-1, 2:m-1]` y `[2:m-1, 1:m-1]`, respectivamente. En la figura 4.5 vemos el código generado para el método `GetCounter(ID)` a partir de esta definición de dependencias del problema Smith-Waterman. Primero calculamos las coordenadas de la tarea a partir de su `ID` para luego en las líneas 5 y 7 comprobar si las coordenadas pertenece a la región `A` y `B`, respectivamente. De esta forma incrementamos el contador una o dos veces dependiendo de si la tarea pertenece a una región o a las dos.

```
1  int Operation::GetCounter(int ID){
2     int i= GetFirstFromID(ID);
3     int j= GetSecondFromID(ID);
4     int counter = 0;
5     if ( ( i>=1 && i<m ) && ( j>=2 && j<m ) )  // A
6        counter++;
7     if ( ( i>=2 && i<m) && ( j>=1 && j<m ) )  // B
8        counter++;
9    return counter;
10 }
```

Figura 4.5: Método `GetCounter()` generado a partir del fichero de definición del problema Smith-Waterman.

Para el cálculo de la dimensión del macrobloque se realizaron experimentos con distintos códigos wavefront en la arquitectura de 32 cores descrita anteriormente. Volviendo al caso del código Smith-Waterman, el tiempo de inicialización sin esta optimización es de 0,12 segundos sobre un tiempo de ejecución de todo el wavefront de 1,6 segundos para una matriz de $40.000 \times 40.000$. Si la matriz es de $10.000 \times 10.000$ el tiempo de

inicialización es de 0,063 sobre un total de 0,9 segundos. Es decir en ambos casos, la inicialización se acerca al 8 % del tiempo total de ejecución.

Inicializando sólo el primer macrobloque antes de disparar la ejecución wavefront, los tiempos de inicialización pueden bajar más de tres órdenes de magnitud. Por ejemplo, 0,29 microsegundos con $mbs = 8$ y 96,3 microsegundos con $mbs = 128$. Note que estos tiempos son independientes del tamaño de la matriz total ya que el resto de los macrobloques de contadores se inicializan de forma solapada con la ejecución del wavefront. Aunque el tiempo de ejecución de una de las tareas de cada macrobloque aumenta en unos microsegundos, el tiempo total de ejecución del wavefront prácticamente no se ve afectado. De esta forma, el porcentaje de tiempo consumido en la inicialización respecto del total es apenas del 0,006 % para la matriz de $40.000 \times 40.000$.

Si queremos que durante la ejecución del primer macrobloque ya estén disponibles suficientes tareas independientes como para ocupar todos los cores disponibles, debemos hacer algún cálculo adicional. En general, nuestros experimentos apuntan a que todos los cores están cerca del 100 % de utilización cuando el número de tareas en vuelo es igual a $1.5$ veces el número de cores. En una máquina de 32 cores, se traduce en 48 tareas independientes. Para conseguir esas tareas paralelas en un problema como Smith-Waterman necesitamos que la antidiagonal del macrobloque tenga ese tamaño o mayor. Concluimos por tanto que tamaños de macrobloque de $64 \times 64$ o $128 \times 128$ son adecuadas en nuestra arquitectura de 32 cores.

## 4.3. Tiling

En este apartado describimos las modificaciones necesarias en la plantilla wavefront para incorporar la técnica de tiling. El tiling consiste en particionar el espacio de datos en bloques regulares. Un caso particular y bien conocido es el de "loop tiling" en el que se particiona el espacio de iteraciones en bloques de forma que los datos del bloque, una vez alojados en cache, se reusan más veces de forma que se aprovecha mejor la localidad y el rendimiento del código aumenta. Nuestra implementación de tiling para wavefront también tendrá efectos beneficiosos, pero más que por un mejor aprovechamiento de la caché, por reducir ciertos overheads de la ejecución. En particular, conseguiremos reducir el número de spawns, el tamaño y el número de accesos a la matriz de contadores y aumentar la granularidad de la tarea.

Hasta ahora, cada celda de la matriz se asociaba a una tarea encargada de llevar a cabo la computación correspondiente a la celda. Nuestro tiling asignará ahora una tarea a cada bloque (o tile) del espacio computacional. Volviendo al problema Smith-Waterman, la figura 4.6, presenta un ejemplo en el que el espacio de los datos es `[0:7,0:7]`, pero

el de tareas es `[1:7,1:7]` y es este último el que se ha particionado en bloques de $2 \times 2$. Por tanto, ahora cada tarea realizará las computaciones asociadas a las cuatro celdas incluidas en cada bloque. El recorrido de las celdas del bloque se realizará de forma secuencial, siguiendo un orden que garantice el respeto a las dependencias entre las celdas del bloque. De esta manera, se logra el objetivo de tener mayor granuralidad de tarea y menos overhead introducido por la gestión de las dependencias (al haber menos tareas en el cómputo total del problema también necesitamos menos contadores atómicos).

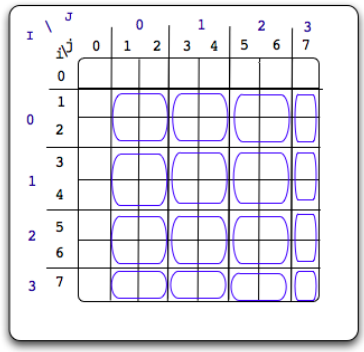



Figura 4.6: Región de bloque en Smith-Waterman.

Desde el punto de vista del usuario, el procedimiento para usar la nueva plantilla apenas cambia. El usuario sigue proporcionando el mismo fichero de definición para indicar el patrón de dependencias, junto con la función `executeTask` que es la que describe la computación a realizar para cada elemento de la matriz. La única diferencia, por ahora, es que al llamar al generador de código que procesa el fichero definición, lo hacemos pasando el argumento `--tile`, que activa las modificaciones necesarias para implementar tiling. Más adelante se comentará otra diferencia respecto al código generado que permite buscar automáticamente el tamaño optimo del tile. Insistimos en que el método `executeTask` es el mismo que ya utilizamos en el capítulo anterior y no necesita ninguna modificación.

Sin embargo, y como es de esperar, desde el punto de vista de la implementación, el fichero de definición ha de ser tratado con consideraciones adicionales para acomodar la técnica de tiling. En particular, habrá que:

- Hacer una conversión de índices de celdas a índices de bloque en las regiones de tareas.

- Llevar a cabo una conversión similar de los vectores de dependencias.
- Evitar que las nuevas dependencias produzcan ciclos.
- Por último, implementar una búsqueda automática que devuelva el tamaño óptimo del tile para el problema wavefront en cuestión.

Pasamos a discutir cada uno de estos puntos en mayor profundidad en las cuatro siguientes sub-secciones.

### 4.3.1. Transformación de regiones

El tiling se puede aplicar a matrices n-dimensionales pero sin perdida de generalidad y dado que la inmensa mayoría de problemas wavefront reales son de tipo 2D, presentamos las transformaciones y los ejemplos para el caso bidimensional. A continuación vamos a exponer unas definiciones y reglas importantes a tener en cuenta al implementar tiling para un fichero de definición genérico como el de la figura 4.7.

```
1 [0:n-1, 0:m-1]     // Data space
2 [li:hi, lj:hj]    // Task space (*@@*)
3 <i,j>             // Variables Índices
4 [lx:hx, ly:hy]-> (dlx:dhx, dly:dhy) // dependencias (*@@*)
```

Figura 4.7: Fichero de definición.

En general, cada bloque está formado por elementos adyacentes que forman una submatriz de $bi * bj$ elementos, donde $bi$ es el número de filas de elementos (coordenada i) y $bj$ el número de columnas de elementos (coordenada j). Cuando $bi = bj$ llamamos $b$ al tamaño de bloque para ambas dimensiones. Dado que el acceso a los datos se sigue realizando en el espacio de iteraciones original, la región de datos no requiere una transformación a índices bloque.

Sin embargo, la región de tareas y las regiones para las que se definen dependencias si habrá que transformarlas al espacio de bloques. Para ello, haremos operaciones de división entre $bi$ y $bj$. Tendremos en cuenta que todas las operaciones de división que realizamos al cambiar de coordendas de celda a coordenadas de bloque son divisiones enteras (*floor* en terminología C). Como vemos en la figura 4.6, utilizaremos $(i, j)$ para referirnos a las coordenadas de las celdas en el espacio original, e $(I, J)$ como coordenadas en el espacio de bloques. La transformación de las coordendas de la celda $(i, j)$ a índices de bloques viene dada por la siguiente ecuación:

$$(I, J) = \left( \left\lfloor \frac{i - li}{bi} \right\rfloor, \left\lfloor \frac{j - lj}{bj} \right\rfloor \right) \tag{4.1}$$

La operación inversa es identificar la región, $[i_1 : i_2, j_1 : j_2]$, de celdas incluidas en el bloque de coordenadas $(I, J)$, para lo cual nos valemos de la siguiente expresión:

$$[i_1 : i_2, j_1 : j_2] = \big[(I{*}bi{+}li) : ((I{+}1){*}bi{+}li{-}1), (J{*}bj{+}lj) : ((J{+}1){*}bj{+}lj{-}1)\big]$$

garantizando que $i_2 = hi$ y que $j_2 = hj$ en caso de que los valores resultantes de los límites superiores superen los límites de la región de tareas, $hi$ y $hj$.

Con esto, la transformación a índices de bloque de una región de celdas se implementa mediante la transformación de cada uno de los límites de la región mediante la ecuación 4.1. Es decir:

$$[lx : hx, ly : hy] \Rightarrow \left[ \left\lfloor \frac{lx - li}{bi} \right\rfloor : \left\lfloor \frac{hx - li}{bi} \right\rfloor, \left\lfloor \frac{ly - li}{bj} \right\rfloor : \left\lfloor \frac{hy - lj}{bj} \right\rfloor \right] \tag{4.2}$$

Por ejemplo, en la figura 4.8 recordamos el fichero de definición original para el problema Smith-Waterman a la izquierda, y a la derecha las regiones que se deben trasformar al espacio de índices de bloques.

Fichero original

```
1 //Data space
2 [0:n-1, 0:n-1]
3 //Task space
4 [1:n-1, 1:n-1]
5 //Indices
6 <i, j>
7 //Dependency vectors
8 [1:n-1, 1:n-1]->(0,1);(1,0)
```

Fichero parcialmente transformado

```
1 //Data space
2 [0:n-1,0:n-1]
3 //Task space
4 [0:(n-2)/bi, 0:(n-2)/bj]
5 //Indices
6 <i, j>
7 //Dependency vectors
8 [0:(n-2)/bi, 0:(n-2)/bj]->(0,1);(1,0)
```

Figura 4.8: Fichero de definición del problema Smith-Waterman en sus versiones original y parcialmente transformada.

### 4.3.2. Transformación de dependencias

En la figura 4.9, recurrimos de nuevo al problema de Smith-Waterman para ilustrar el proceso de transformación de dependencias al considerar tiling como estrategia de optimización. En dicha figura distinguimos las dependencias "intra-tile", en las que una celda depende de otra del mismo tile, y las dependencias "inter-tile", en las que el vector de dependencia cruza la frontera del tile.

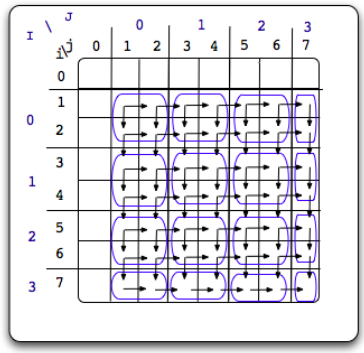



Figura 4.9: Patrón de dependencias en el problema Smith-Waterman con tiling.

Para gestionar las dependencias intra-tile nos limitamos a recorrer cada tile de forma secuencial siguiendo un orden que respete dichas dependencias. En particular, el bucle secuencial que recorre todas las celdas del tile seguirá la dirección principal proyectada sobre el eje de abscisas. De esta forma priorizamos el recorrido por filas ya que en C, ese recorrido se ajusta al orden en el que se almacenan los datos en memoria y así aprovechamos mejor la localidad espacial. Por ejemplo, en el código Smith-Waterman en cuestión, un doble bucle anidado, con el recorrido de cada fila en el bucle interno, será generado automáticamente. En el cuerpo de ese anidamiento sólo hay que llamar a la función `executeTask` que lleva a cabo la computación para cada celda.

Por otro lado, las dependencias inter-tile son las que nos permiten encontrar las dependencias que ahora determinan cómo inicializar los contadores atómicos (ahora uno por cada tile) y qué tiles hay que despertar cuando uno se completa. Dado que todas las celdas dentro de un tile se ejecutan de forma monolítica, no podemos comenzar el procesado de dicho tile hasta que todas las dependencias con celdas de fuera del tile se hayan satisfecho. Por tanto el problema a resolver aquí es el de encontrar, a partir del patrón de dependencias de celdas, el patrón equivalente de dependencias entre tiles.

El proceso para encontrar a qué bloques despierta uno dado, consiste en, i) aplicar las dependencias a todas las celdas del bloque, lo que resulta en una región de celdas dependientes; ii) determinar en que bloques se alojan dichas celdas dependientes; y iii) calcular los vectores, en el espacio de bloques que apuntan a estos bloques. Este proceso es equivalente a convertir los vectores de dependencias, aplicados a las cuatro esquinas de cada bloque, a coordenadas de bloque. Es decir, dado un vector de dependencias $(dx, dy)$ y un tamaño de bloque $(bi, bj)$, los desplazamientos relativos a las coordenadas de todas las celdas de un bloque genérico son:

$$(dx : dx + bi - 1, dy : dy + bj - 1)$$

que convertido a coordenadas de bloque resulta en:

$$\left(\left\lfloor \frac{dx}{bi} \right\rfloor : \left\lfloor \frac{dx + bi - 1}{bi} \right\rfloor, \left\lfloor \frac{dy}{bj} \right\rfloor : \left\lfloor \frac{dy + bj - 1}{bj} \right\rfloor\right) \tag{4.3}$$

Por tanto, para un vector de dependencias genérico, `(dlx:dhx, dly:dhy)`, habría que aplicar la ecuación 4.3 a los cuatro vectores simples resultantes: `(dlx,dly)`, `(dlx,dhy)`, `(dhx,dly)`, `(dhx,dhy)`. Tras esta conversión pueden aparecer dependencias de tipo $(0, 0)$ (intra-bloque), que hay que eliminar, o dependencias repetidas de las que hay que mantener una sola instancia.

Para el problema Smith-Waterman, el descriptor del patrón de dependencias viene dado por `[0:(n-2), 0:(n-2)]->(0,1);(1,0)`, que transformado a coordenadas de bloques, tanto en la región como en las dependencias, queda como vemos en la figura 4.10, si consideramos un tamaño de bloque de dimensiones $bi \times bj$.

```
1 [0:(n-2)/bi, 0:(n-2)/bj] -> (0:(bi-1)/bi, 1/bj:(1+bj-1)/bj);
2                             (1/bi:(1+bi-1)/bi, 0:(bj-1)/bj)
```

Figura 4.10: Transformación de las dependencias en el problema Smith-Waterman.

Como vemos, para el caso sin tiling en que $bi = 1$ y $bj = 1$, la expresión degenera de nuevo en `[0:(n-2), 0:(n-2)]->(0,1);(1,0)`. Si $bi > 1$ y $bj > 1$, la expresión también se puede simplificar y los vectores quedarían del tipo `(0:0, 0:1);(0:1, 0:0)`. Teniendo en cuenta que el vector (0,0) no representa realmente una dependencia, la expresión final sería `[0:(n-2)/bi, 0:(n-2)/bj]->(0,1);(1,0)`.

Por otro lado, aunque tan solo ocurre en un problema real de tipo wavefront que conozcamos (Floyd, que por su naturaleza no mejora haciendo tiling), el fichero de definición permite declarar distintos patrones de dependencias para distintas regiones. Esta

flexibilidad da lugar a que al hacer el tiling del problema aparezcan bloques que pertenecen a varias regiones con patrones de dependencias diferentes. Por ejemplo, en la figura 4.11, vemos las regiones `A=[1:7,1:3]` y `B=[1:7,4:7]`, y que al aplicar tiles de $2 \times 2$ nos quedan unas regiones de bloque `A'=[0:3,0:1]` y `B'=[0:3,1:3]`, donde los tiles `[0:3,1]` están sujetos a los dos patrones de dependencias.

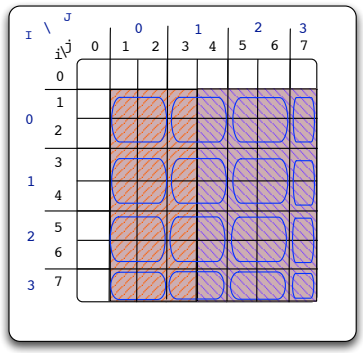



Figura 4.11: Ejemplo de regiones solapadas al hacer tiling.

Esta situación está gestionada internamente ya que la plantilla soporta que una celda pertenezca a varias regiones. La implementación chequea para cada celda a cuantas regiones pertenece y aplica las dependencias correspondientes. La forma de proceder cuando se hace tiling y aparecen bloques pertenecientes a distintas regiones es similar, con la salvedad de que hay que comprobar que no aparezcan ciclos de dependencias. Sin tiling, es responsabilidad del usuario evitar ciclos ya que pueden dar lugar a interbloqueos, pero dado que el tiling es automático, la tarea de comprobar que la ejecución wavefront esta libre de interbloqueos recae ahora en la implementación de la plantilla. A continuación abordamos este aspecto.

### 4.3.3. Gestión de ciclos de dependencias

Intuitivamente, un ciclo de dependencias se produce cuando dos celdas tienen dependencias entre si. Teniendo en cuenta la transitividad entre dependencias, esto ocurre no sólo cuando la celda A depende de la celda B y viceversa, sino también cuando entre A y B hay otras celdas en la secuencia de dependencias. Más formalmente, dada una celda $X$ de coordenadas $(i, j)$ perteneciente a una región $R$ en la que se definen un conjunto de dependencias $d = \{d_1, d_2, ..., d_n\}$, decimos que aparece un ciclo cuando

aplicando a $X$ una o más dependencias, una o más veces, alcanzamos de nuevo las coordenadas $(i, j)$ de la celda $X$. Esto es generalizable al caso en que haya varias regiones, $R^1, R^2, ..., R^m$, alcanzables desde una celda $X$ que pertenece a una de esas regiones, en cuyo caso habrá que considerar todas los conjuntos de dependencias, $d^1, d^2, ..., d^n$, siendo $d^i = \{d^i_1, d^i_2, ..., d^i_n\}$. Sin embargo, dado que esto no ocurre en problemas reales de tipo wavefront y que la explicación se simplifica considerando una única región, $R$, sin perdida de generalidad, en lo que sigue nos atendremos a este caso.

Esta situación se puede ilustrar usando el patrón de dependencias del problema Checkerboard. En la figura 4.12 vemos el patrón de dependencias de este problema, `[1:m-1, 0:n-1]->(1,-1);(1,0);(1,1)`, y el resultado de aplicar tiling para dos tamaños diferentes de tile: $2 \times 2$ a la izquierda y $1 \times 2$ a la derecha. Para el primer caso, las flechas rojas ponen de manifiesto como dos bloques consecutivos de la misma fila dependen entre sí. Sin embargo esta situación no se da cuando la dimensión elegida para el tile es $1 \times 2$. Es responsabilidad de la plantilla avisar al usuario cuando éste intente configurar una dimensión de tile que da lugar a ciclos. Además, como veremos más adelante, si damos soporte a la búsqueda automática del tamaño óptimo del tile, debemos ser capaces de eliminar de dicha búsqueda aquellas dimensiones que provoquen ciclos.

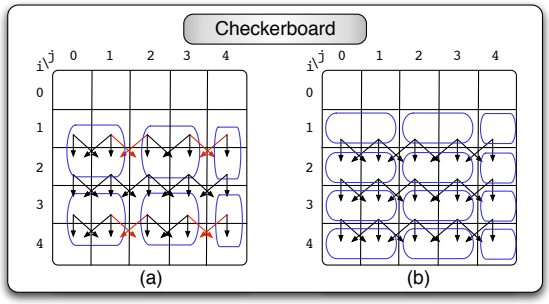



Figura 4.12: Tiling en el problema del Checkerboard. Aparecen ciclos si la dimensión del tile es $2 \times 2$, pero no es el caso si el tile es de $1 \times 2$.

Pasamos entonces a describir la solución que permite detectar ciclos automáticamente. Como hemos descrito anteriormente, tenemos un ciclo cuando a partir de una celda de una región, $R$, en la que se definen un conjunto de $d$ dependencias, $d = \{d_1, d_2, ..., d_d\}$, aplicando una o más dependencias, una o más veces, alcanzamos de nuevo ese punto. Es decir, existen $x_1, x_2, x_3, ..., x_d$ enteros positivos, con al menos uno de ellos mayores que 0, tal que existe una combinación lineal de dependencias que resulta en el vector de

dependencias neutro (es decir, la dependencia $(0, 0)$ en un espacio 2D).

$$x_1 * d_1 + x_2 * d_2 + \cdots + x_d * d_d = d_e \tag{4.4}$$

donde, $d_e$ es un vector de dependencia neutro y al menos $x_i$, es mayor que cero. En un espacio 2D, $d_e = (0, 0)$ y tenemos dos dimensiones por dependencia, $(dx, dy)$, así que podemos reescribir la combinación lineal como un sistema de ecuaciones con restricciones, tal y como muestran las ecuaciones 4.5 a 4.7.

$$x_1 * dx_1 + x_2 * dx_2 + \cdots + x_d * dx_d = 0; \qquad (eq1) \tag{4.5}$$

$$x_1 * dy_1 + x_2 * dy_2 + \cdots + x_d * dy_d = 0; \qquad (eq2) \tag{4.6}$$

$$x_1, x_2, ..., x_d \geq 0, \text{ con } x_i > 0, \tag{4.7}$$

Para saber si hay solución en el sistema, utilizamos el I-test[44], el cual nos dice si hay solución entera para las ecuaciones $(eq1)$ y $(eq2)$, permitiendo especificar algunas restricciones para cada variable: $M_k \leq x_k \leq N_k$, donde $M_k, N_k$ son enteros. El I-test puede devolver tres resultados distintos: "no" (no hay solución), "yes" (hay solución) y "maybe" (puede haber solución). Para detectar la presencia de un ciclo de dependencias para un tamaño de bloque dado, debemos aplicar el I-test para ambas ecuaciones por separado. Si el I-test devuelve *no* para alguna de las dos ecuaciones (eq1 o eq2), podemos decir que no existe ningún ciclo. En otro caso, no podemos afirmar nada, pero conservativamente decidimos no utilizar ese tamaño de bloque ya que es sospechoso de provocar un ciclo.

Más precisamente, nuestra aproximación descompone la ecuación con la combinación lineal de los vectores de dependencias en tantas ecuaciones como dimensiones tengan los vectores. En nuestro caso, cuando la ecuación 4.4 contiene vectores de dos dimensiones, la descomponemos en las ecuaciones $(eq1)$ y $(eq2)$. La ecuación 4.4 tiene solución entera si se encuentran los valores $x_1, x_2, \cdots, x_d$ que la satisfacen. Nosotros somos capaces de encontrar las situaciones en que $(eq1)$ y $(eq2)$ tienen solución entera, pero no buscamos los valores específicos de $x_1, x_2, \cdots, x_d$ que satisfacen cada ecuación y tampoco chequeamos que esos coeficientes satisfacen simultáneamente $(eq1)$ y $(eq2)$. En otras palabras, nuestra estrategia puede dar falsos positivos no sólo porque el I-test pude dar un falso positivo devolviendo "maybe", sino porque consideramos un ciclo potencial cuando se devuelve "yes" para las dos ecuaciones, aún sin comprobar que sea exactamente la misma solución para las dos.

Para estudiar cada ecuación con el I-test, necesitamos definir las restricciones para cada variable. La primera restricción es el límite inferior, $M_k$, para cada $x_k$. Este, en

principio, es 0 porque las dependencias se aplican 0 o más veces (un número entero positivo de veces). El límite superior, $N_k$, es $m$ o $n$ (si $m \times n$ es la dimensión de la matriz). Es decir, para la primera ecuación del sistema, $(eq1)$, estamos considerando la componente $dx$ del vector que recorre las filas de la matriz, y si sólo hay $m$ filas no podemos aplicar la dependencia más de $m$ veces, es decir $N_k = m$. De forma similar, para la segunda ecuación del sistema, $(eq2)$, usaremos $N_k = n$. La otra restricción es que necesitamos combinar al menos dos dependencias para generar el ciclo. Esto es trivial ya que teniendo en cuenta que no existe la dependencia $(0, 0)$ en el conjunto, al menos dos coeficientes deben ser mayor que 0 para obtener el elemento neutro por combinación lineal. Sin embargo, basta con fijar una de las dependencias mayores que 0 ($x_i > 0$ ya que si el I-test devuelve cierto, en este caso, significa que existirá otra dependencia mayor que 0. Esto implica $M_i = 1$.

Por otro lado, si consideramos el tamaño de bloque, $bi \times bj$, los límites superiores $N_k$ se pueden hacer más restrictivos aún. Si la dimensión de la matriz es de $m \times n$, realmente tenemos $\lceil m/bi \rceil$ bloques en la dimensión $i$, por lo que ese será el número máximo de veces que puedes aplicar la componente $dx$ de los vectores de dependencia para la $(eq1)$. Así, realmente $N_k = \lceil m/bi \rceil$. De forma similar, para la $(eq2)$ tendremos $N_k = \lceil n/bj \rceil$. De cualquier modo, si no encontramos solución con un valor alto de $N_k$, podemos estar seguros de que no lo vamos a encontrar tampoco con uno más pequeño. Es decir, si no encontramos una solución entera cuando las variables pueden moverse en el rango $[M_k, N_k]$, menos la vamos a encontrar si ahora el rango de la variable es menor, $[M_k, N_k/b]$.

Para entender mejor como particularizamos el I-test para resolver nuestro problema de detección de ciclos, resumimos a continuación los aspectos más importantes de este algoritmo.

#### 4.3.3.1. I-test

El I-test [44] está basado en el test GCD y en el test de Banerjee. Estos tres tests fueron ideados inicialmente en el campo de paralelización y vectorización automática. Básicamente, estos test resuelven el problema de "resolución de alias en un array": determinar si dos referencias al mismo array dentro de un anidamiento de bucles pueden acceder a un mismo elemento del array. Cuando el array referenciado es de $m$ dimensiones y los índices del array son lineales, el problema se reduce a averiguar si un sistema de $m$ ecuaciones lineales, cada una de la forma

$$a_1 * I_1 + a_2 * I_2 + \cdots + a_n * I_n = a_0 \tag{4.8}$$

tiene una solución entera simultánea que satisface las restricciones de la forma:

$$M_k \leq a_k \leq N_k, \quad 1 \leq k \leq n$$

Cada $I_k$ es una variable de inducción del bucle u otra variable del código. Por tanto, en muchos casos los límites en los que se mueve la variable, $M_k$ y $N_k$, se pueden conocer. El I-test complementa y mejora los tests GCD [5] y Banerjee [5], ya que GCD encuentra si hay soluciones enteras al problema pero no tiene en cuenta los límites de las variables y Banerjee si tiene en cuenta los límites pero informa de si existe una solución real (no entera) al problema. Por tanto, GCD puede dar falsos positivos si encuentra una solución entera al problema fuera de los límites, mientras que Banerjee da falsos positivos si encuentra soluciones reales dentro de los límites. El I-test resuelve estos problemas informando sólo de soluciones enteras dentro de los límites.

En primer lugar, recordamos que el GCD se basa en un teorema elemental de la teoría de números que enuncia que si el gcd($a_1, a_2, ..., a_n$) es un divisor de $a_0$ la ecuación 4.8 tiene una solución entera. Por tanto el test consiste en chequear dicha divisibilidad. Por su parte, el test de Banerjee considera los límites de las variables sólo si estos tienen un valor constante determinado, ya que en otro caso no es aplicable. Para el test de Banerjee se usa la siguiente definición.

**Definición 4.1.** *Sea $a$ un entero,*

$$a^+ = a \qquad \textit{si } a \geq 0, \quad 0 \textit{ en otro caso}$$
$$a^- = -a \qquad \textit{si } a \leq 0, \quad 0 \textit{ en otro caso}$$

Además, cuando el test de Banerjee es aplicable, primero se calculan los valores extremos

$$\mathit{low} = \sum_{i=1}^{n} (a_i^+ M_i - a_i^- N_i)$$

y

$$\mathit{high} = \sum_{i=1}^{n} (a_i^+ N_i - a_i^- M_i)$$

con *low*<*high*. Por el Teorema del Valor Intermedio, la ecuación 4.8, tiene una solución real entre los límites si $\mathit{low} < a_0 < \mathit{high}$. El test de Banerjee se basa en chequear dicha condición.

El I-test es un refinamiento de la combinación del test GCD y Banerjee, que, al igual que GCD, chequea la existencia de soluciones enteras, y al igual que Banerjee, tiene los límites en cuenta. Para discutir la implementación del I-test, enunciamos primero dos definiciones.

**Definición 4.2.** *Solubilidad: Sean $a_0, a_1, a_2, \cdots, a_n$ enteros. Para cada k, $1 \leq k \leq n$, sean cada uno de los $M_k$ y $N_k$ enteros o límites desconocidos, lo cual se indica mediante el símbolo "$*$", donde $M_k \leq N_k$ si ambos $M_k$ y $N_k$ son enteros.*

*Si $n > 0$, entonces la ecuación:*

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = a_0$$

*es $(M_1, N_1; M_2, N_2; \cdots; M_n, N_n)$-entero-soluble, si existen enteros $j_1, j_2, \cdots, j_n$ tales que*

- $a_i j_1 + a_2 j_2 + \cdots + a_n j_n = a_0$
- *para cada $k, 1 \leq k \leq n$:*
  - *Si $M_k$ y $N_k$ son enteros, $M_k \leq j_k \leq N_k$.*
  - *Si $M_k$ es un entero, y $N_k = *$, entonces $M_k \leq j_k$*
  - *Si $M_k = *$ y $N_k$ es entero, entonces $j_k \leq N_k$*

*Si $n = 0$, la ecuación*

$$\begin{aligned} a_i j_1 + \cdots + a_n j_n &= a_0 \\ \equiv 0 &= a_0 \end{aligned}$$

*es entero soluble si $a_0 = 0$.*

**Definición 4.3.** *Ecuación de Intervalos: El I-test trata con una ecuación transformándola en una ecuación de intervalos. Sean $a_1, a_2, \cdots, a_n, L$ y $U$ enteros. La ecuación:*

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = [L, U],$$

*a la que nos referiremos como ecuación de intervalos, será usada para representar un conjunto de ecuaciones consistentes en:*

$$\begin{aligned} a_1 I_1 + a_2 I_2 + \cdots + a_n I_n &= L \\ a_1 I_1 + a_2 I_2 + \cdots + a_n I_n &= L + 1 \\ &\vdots \\ a_1 I_1 + a_2 I_2 + \cdots + a_n I_n &= U \end{aligned}$$

Además, el I-test enuncia y demuestra 7 teoremas en el artículo publicado en 1991, pero para entender su uso para nuestro problema particular, en el que es posible aplicar ciertas simplificaciones, basta con que recordemos aquí tres de ellos. Las demostraciones de estos teoremas están a disposición del lector en [44].

**Teorema 4.1.** *Test GCD de ecuación de intervalo: Sean $a_1, a_2, \cdots, a_n, L$, y $U$ enteros, y sea $d = gcd(a_1, a_2, \cdots, a_n)$, la ecuación de intervalo*

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = [L, U]$$

*tiene solución entera si y sólo si $L \leq d\lceil L/d \rceil \leq U$.*

**Teorema 4.2.** *Sean $a_1, a_2, \cdots, a_n, L$, y $U$ enteros. Para cada $k, 1 \leq k \leq n-1$, sean $M_k, N_k$ enteros o el valor desconocido "*", donde $M_k \leq N_k$ si $M_k$ y $N_k$ son enteros. Sean $M_n, N_n$ enteros y $M_n \leq N_n$. Si $|a_n| \leq U - L + 1$, entonces, la ecuación de intervalos*

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = [L, U]$$

*es $(M_1, N_1; M_2, N_2; \cdots; M_n, N_n)$-entero-soluble si y sólo si la ecuación de intervalos*

$$a_1 I_1 + a_2 I_2 + \cdots + a_{n-1} I_{n-1} = [L - a_n^+ N_n + a_n^- M_n, U - a_n^+ M_n + a_n^- N_n]$$

*es $(M_1, N_1; M_2, N_2; \cdots; M_{n-1}, N_{n-1})$-entero-soluble.*

**Ejemplo 4.1.** *Para ilustrar cómo podemos utilizar estos dos últimos teoremas para mejorar la precisión del test GCD y del test de Banerjee, consideramos la ecuación:*

$$I_1 - 3I_2 + 7I_3 = 8$$

*sujeto a los límites*

$$1 \leq I_1 \leq 3, \quad 1 \leq I_2 \leq 2, \quad 1 \leq I_3 \leq 4$$

*En este caso $gcd(1, -3, 7) = 1$, que es divisor de 8, así que el test GCD indica que la ecuación puede ser (1,3; 1,2; 1,4)-entero-soluble. Por su parte el test de Banerjee calcula los valores extremos para la expresión $I_1 - 3I_2 + 7I_3$ dentro de los límites de $I_1$, $I_2$ e $I_3$, que resultan en low= 2 y high= 28. Dado que $2 \leq 8 \leq 28$, el test Banerjee también indica que puede existir una solución para nuestra ecuación.*

*Sin embargo, si reescribimos la ecuación como*

$$I_1 - 3I_2 + 7I_3 = [8, 8]$$

*y aplicamos el teorema 2.2, vemos que el coeficiente que cumple que* $|a_i| \leq U - L + 1 = 1$ *es el primero, con límites,* $M_1 = 1$ *y* $N_1 = 3$*. Eliminando ese coeficiente obtenemos:*

$$-3I_2 + 7I_3 = [8 - 3, 8 - 1] = [5, 7]$$

*que según el test GCD de la ecuación de intervalo aún indica que puede existir una solución. Pero aún existe un coeficiente que cumple* $|a_i| \leq 7 - 5 + 1 = 3$*, así que podemos volver a aplicar el teorema 2.2, esta vez moviendo el termino* $-3I_2$ *al lado derecho. Ahora los límites son,* $M_2 = 1$ *y* $N_2 = 2$ *y* $a_2^+ = 0$*, pero* $a_2^- = 3$*, de forma que obtenemos:*

$$7I_3 = [5 + 3, 7 + 6] = [8, 13]$$

*Ya no quedan coeficientes que cumplan que* $|a_i| \leq 13 - 8 + 1 = 6$*, pero sigue quedando un coeficiente con* $d = gcd(7) = 7$*, y aplicando el teorema 2.1 o test GCD de la ecuación de intervalos se concluye que no existe solución entera para este caso ya que no se cumple que* $L \leq d\lceil L/d \rceil \leq U$ *(en este caso* $7 \times \lceil 8/7 \rceil = 7 \times 2 = 14 > 13$*).*

**Teorema 4.3.** *Sean* $a_1, a_2, \cdots, a_n, L$*, y* $U$ *enteros. Para cada* $k, 1 \leq k \leq n$*, sea* $M_k, N_k$ *enteros o el valor desconocido "*", donde* $M_k \leq N_k$*, si* $M_k, N_k$ *son enteros. Sea* $d = gcd(a_1, a_2, \cdots, a_n)$ *y la ecuación de intervalos*

$$a_1 I_1 + a_2 I_2 + \cdots + a_n I_n = [L, U]$$

*es* $(M_1, N_1; M_2, N_2; \cdots; M_n, N_n)$*-entero-soluble si y sólo si la ecuación de intervalos*

$$(a_1/d)I_1 + (a_2/d)I_2 + \cdots + (a_n/d)I_n = [\lceil L/d \rceil, \lfloor U/d \rfloor] \tag{4.9}$$

*es* $(M_1, N_1; M_2, N_2; \cdots; M_n, N_n)$*-entero-soluble.*

*Nota: la ecuación 4.9 tiene coeficientes enteros porque* $d = gcd(a_1, a_2, \cdots, a_n)$*.*

**Ejemplo 42.** *Para ilustrar en qué medida puede ser útil el teorema 2.3, consideremos la siguiente ecuación:*

$$2I_1 - 6I_2 + 14I_3 = [16, 16]$$

*sujeto a los límites* $1 \leq I_1 \leq 3, 1 \leq I_2 \leq 2, 1 \leq I_3 \leq 4$*, y a la que no se puede aplicar el teorema 2.2 ya que no tenemos coeficientes lo suficientemente pequeños. Sin embargo, si aplicamos el teorema 2.3, con* $d = gcd(2, 6, 14) = 2$ *obtenemos* $I_1 - 3I_2 + 7I_3 = [8, 8]$*, que ya vimos en el ejemplo 2.2 que no tiene solución entera en los límites considerados.*

```
1  input: (a_o, a_1, ..., a_n, M_1, N_1, ..., M_n, N_n)
2  output:  no, yes, o maybe
3  Algoritmo:
4   L = a_0
5   U = a_0
6   coeff = {a_1, a_2, ..., a_n}
7   unknown = {a_i ∈ coeff |M_i = * y  N_i = *}
8   if (unknown ≠ ∅){
9     u =     gcd   (a)
            a∈unknown
10    if (u=1) return(yes)
11    coeff = (coeff - unknown) ∪ {u}
12  }
13  while(true){
14    while (∃a_i ∈ coeff tal que |a_i| ≤ U-L+1 y  M_i ≠ *, N_i ≠ *){
15      /* Teorema 2.2 */
16      L = L - a_i^+ N_i + a_i^- M_i
17      U = U - a_i^+ M_i + a_i^- N_i
18      coeff = coeff -{a_i}
19      if (coeff = ∅)
20        if (L ≤ 0 y 0 ≤ U)
21          return(yes)
22        else
23          return(no)
24     }
25     d =     gcd   (a)
            a∈coeff
26     /* Teorema 2.1. Chequea el test GCD de la ec. de intervalo */
27     if ( not(L ≤ d⌈L/d⌉ ≤ U) ) return(no)
28
29    /* Teorema 2.3 */
30     if (d ≠ 1){
31       for all a ∈ coeff a = a/d
32       L = ⌈L/d⌉
33       U = ⌊U/d⌋
34     }
35     else return(maybe)
36  }
```

Figura 4.13: Pseudocódigo del algoritmo I-test.

Con todo esto podemos describir el algoritmo que implementa el I-test en la figura 4.13. En la línea 1 se especifica la entrada al algoritmo: los coeficientes y los límites inferior y superior de cada variable. A la salida podemos obtener que “no” existe solución entera, que “si” o que “puede ser”. En las líneas 4 y 5 se asignan los valores iniciales para L y U. Las líneas entre 7 y 12 son útiles cuando no se conocen algunos de los límites de las variables, pero dado que eso no ocurre en nuestro problema de detec-

ción de ciclos no discutiremos aquí esa posibilidad. En la línea 14 se muestra el bucle que itera mientras haya algún coeficiente ($a_i$) en *coeff* tal que $|a_i| \leq U - L + 1$, es decir, para los coeficientes que satisfacen el teorema 2.2. Según este teorema, en este bucle se actualizan los valores de $L$ y $U$ en las líneas 16 y 17 y eliminamos cada coeficiente que ha sido movido a la derecha. Si no quedan coeficientes (línea 19), devolvemos "yes" cuando $L \leq 0$ y $0 \leq U$ y "no" en caso contrario. Si aún quedan coeficientes seguiremos intentando aplicar el teorema 2.2 hasta que no queden coeficientes o los que queden no cumplan $|a_i| \leq U - L + 1$. En este último caso salimos del bucle y en la línea 25 calculamos el $\gcd$ de todos los coeficientes restantes para intentar aplicar el teorema 2.1. Si se cumple (not($L \leq d\lceil L/d \rceil \leq U$)), el algoritmo devuelve no. En otro caso, si $\gcd \neq 1$ podemos aplicar el teorema 2.3, actualizamos los coeficientes, línea 31, así como $L$ y $U$ (líneas 32 y 33) para volver otra vez el bucle de la línea 14. Sin embargo, si $\gcd = 1$, el algoritmo termina devolviendo "maybe".

```
1  input: (dx_1, dx_2, ..., dx_n, M_1, N_1, ..., M_n, N_n)
2  output:  no, yes, o maybe
3  Algoritmo:
4   L=U=0
5   coeff = {dx_1, dx_2, ..., dx_n}
6   while(true){
7     while (∃dx_i ∈ coeff tal que |dx_i| ≤ U − L + 1){
8       /* Teorema 2.2 */
9       L = L − dx_i^+ N_i + dx_i^− M_i
10      U = U − dx_i^+ M_i + dx_i^− N_i
11      coeff = coeff −{dx_i}
12      if (coeff = ∅)
13        if (L ≤ 0 y 0 ≤ U)
14          return(yes)
15        else
16          return(no)
17    }
18    d =  gcd (a)
           a∈coeff
19    /* Teorema 2.1. Chequea el test GCD de la ec. de intervalo */
20    if ( not(L ≤ d⌈L/d⌉ ≤ U) ) return(no)
21   /* Teorema 2.3 */
22    if (d ≠ 1){
23      for all a ∈ coeff a = a/d
24      L = ⌈L/d⌉
25      U = ⌊U/d⌋
26    }
27    else return(maybe)
28  }
```

Figura 4.14: Pseudocódigo del algoritmo de detección de ciclos.

#### 4.3.3.2. Adaptación del I-test para detección de ciclos

El algoritmo descrito se puede simplificar ligeramente teniendo en cuenta que sólo lo vamos a aplicar para nuestro caso particular de detectar qué tamaños de bloque no provocan ciclos cuando aplicamos tiling. El código simplificado se muestra en la figura 4.14. Las consideraciones particulares a nuestro problema son las siguientes. En primer lugar, todos los límites $M_k, N_k$ son conocidos por lo que podemos eliminar las líneas 7 y 12 del algoritmo anterior. Por otro lado, $a_0 = 0$ y en nuestro problema los coeficientes son los vectores de dependencias, $dx_i$, mientras que buscamos si existen los enteros $x_i$ que den solución a la ecuación 4.5.

#### 4.3.3.3. Ejemplo: análisis de ciclos para el problema Checkerboard

Para ilustrar el funcionamiento de este algoritmo realizamos el estudio de la existencia de ciclo para el problema Checkerboard. Recordemos que el patrón original de dependencias es (1,-1); (1,0); (1,1). Por tanto, transformando estos vectores mediante la ecuación 4.3 y un tamaño genérico de bloques $bi \times bj$, obtenemos el vector de dependencias en coordenadas de bloques:

$$(1,-1) \Rightarrow \left( \left\lfloor \frac{1}{bi} \right\rfloor : \left\lfloor \frac{1+bi-1}{bi} \right\rfloor, \left\lfloor \frac{-1}{bj} \right\rfloor : \left\lfloor \frac{-1+bj-1}{bj} \right\rfloor \right)$$
$$(1,0) \Rightarrow \left( \left\lfloor \frac{1}{bi} \right\rfloor : \left\lfloor \frac{1+bi-1}{bi} \right\rfloor, \left\lfloor \frac{0}{bj} \right\rfloor : \left\lfloor \frac{0+bj-1}{bj} \right\rfloor \right)$$
$$(1,1) \Rightarrow \left( \left\lfloor \frac{1}{bi} \right\rfloor : \left\lfloor \frac{1+bi-1}{bi} \right\rfloor, \left\lfloor \frac{1}{bj} \right\rfloor : \left\lfloor \frac{1+bj-1}{bj} \right\rfloor \right)$$

**Ejemplo 43.** *Si por ejemplo seleccionamos un tamaño de bloque de* $1 \times 2$*, esas dependencias se simplifican en los vectores* `(1:1, -1:0); (1:1, 0:0); (1:1, 0:1)`*. Hacemos notar, que para el primer vector, la operación* $\lfloor -1/2 \rfloor = -1$*. Si desenrollamos los vectores y eliminamos los vectores repetidos y el vector nulo, (0,0), resultan los tres vectores originales:* `(1,-1); (1,0); (1,1)`*. Por tanto, la ecuación que necesitamos estudiar en este caso para comprobar la existencia de ciclos es:*

$$x_1(1,-1) + x_2(1,0) + x_3(1,1) = (0,0)$$

*que descomponemos en*

$$x_1 + x_2 + x_3 = 0; \qquad (eq1)$$
$$-x_1 + x_3 = 0; \qquad (eq2)$$
$$x_1, x_2, x_3 \geq 0, \text{ con } x_i > 0.$$

*La última restricción no sólo implica que los vectores se deben aplicar 0 o más veces, sino que como mínimo un vector se debe aplicar al menos una vez (teniendo en cuenta que la dependencia (0,0) no existe, para que el I-test devuelva cierto, sería necesario que otro $x_j$ se aplique también una vez). Si además tenemos en cuenta que la región de tareas es* `[1:m-1, 0:n-1]`*, los valores para los límites, $M_k, N_k$, serán:*

- *Para la $(eq1)$ en la que nos movemos en la coordenada $i$: $\forall k, N_k = m$ y $M_k = 0$ salvo para dos coeficientes $M_i$.*
- *Para la $(eq2)$ en la que nos movemos en la coordenada $j$: $\forall k, N_k = n$ y $M_k = 0$ salvo para dos coeficientes $M_i = 1$.*

*Iremos iterando sobre las dependencias para establecer los límites inferiores que deben ser iguales a 1: $M_1 = 1$, $M_2 = 1$ y $M_3 = 1$. Ilustramos el primer caso para la ecuación $(eq1)$ siguiendo el pseudocódigo de nuestro algoritmo de detección de ciclos de la figura 4.14,*

1. $x_1 + x_2 + x_3 = [0, 0]$ *(línea 4)*
2. $x_2 + x_3 = [-m, -1]$ *(líneas 9 y 10)*
3. $x_3 = [-2m, -1]$ *(líneas 9 y 10)*
4. $[L, U] = [-3m, -1]$ *y con $L \leq 0$ y $U < 0$ devuelve "**no**"; (línea 13)*

*Claramente, ese es el mismo resultado que obtenemos para los otros dos casos ($M_2 = 1$ y $M_3 = 1$). Si una ecuación no tiene solución, no hace falta evaluar las demás (en este caso $(eq2)$) ya que podemos garantizar que no existe solución para el sistema. Por tanto, para este tamaño de bloque podemos asegurar que no hay bucles, como se aprecia en la figura 4.12(b).*

**Ejemplo 44.** *Sin embargo, si probamos un tamaño de bloque $2 \times 2$ ($bi = 2$ y $bj = 2$), las dependencias transformadas a coordenadas de bloque son* `(0:1, -1:0); (0:1, 0:0); (0:1, 0:1)`*. Desenrollando los vectores y eliminando los vectores repetidos y el vector nulo, resultan los tres vectores originales más vectores más:*

`(1,-1); (1,0); (1,1); (0,-1); (0,1)`. *Por tanto, la ecuación que necesitamos estudiar en este caso para comprobar la existencia de ciclos es:*

$$x_1(1,-1) + x_2(1,0) + x_3(1,1) + x_4(0,-1) + x_5(0,1) = (0,0)$$

*que separando las dimensiones podemos escribir así:*

$$\begin{aligned} x_1 + x_2 + x_3 &= 0; && (eq1)\\ -x_1 + x_3 - x_4 + x_5 &= 0; && (eq2)\\ x_1, x_2, x_3, x_4, x_5 &\geq 0, \text{ con } x_i, x_j > 0, i \neq j. \end{aligned}$$

*Para el caso:* $M_1 = M_2 = M_3 = M_5 = 0$, $M_4 = 1$ *mostramos la evolución del algoritmo para la ecuación* $(eq1)$*:*

1. $x_1 + x_2 + x_3 = [0, 0]$ *(línea 4)*
2. $x_2 + x_3 = [-m, 0]$ *(líneas 9 y 10)*
3. $x_3 = [-2m, 0]$ *(líneas 9 y 10)*
4. $[L, U] = [-3m, 0]$ *y con* $L \leq 0$ *y* $0 \leq U$ *devuelve "**yes**"; (línea 13)*

*De forma similar, para la ecuación* $(eq2)$*:*

1. $-x_1 + x_3 - x_4 + x_5 = [0, 0]$ *(línea 4)*
2. $x_3 - x_4 + x_5 = [0, n]$ *(líneas 9 y 10)*
3. $-x_4 + x_5 = [-n, n]$ *(líneas 9 y 10)*
4. $x_5 = [-n + 1, 2n]$ *(líneas 9 y 10)*
5. $[L, U] = [-2n + 1, 2n]$ *y dado que* $n$ *tiene que ser mayor que 0, se cumple que* $L \leq 0$ *y* $0 \leq U$ *y devuelve "**yes**"; (línea 13)*

*Dado que existe una solución entera para las dos ecuaciones, conservativamente eliminamos* $2 \times 2$ *como un posible tamaño de bloque ya que puede provocar ciclo. En este caso no estamos ante un falso positivo, ya que los coeficientes* $x_1 = x_2 = x_3 = x_5 = 0, x_4 = 1$ *satisfacen simultáneamente* $(eq1)$ *y* $(eq2)$*, y por tanto la ecuación original*

$$x_1(1,-1) + x_2(1,0) + x_3(1,1) + x_4(0,-1) + x_5(0,1) = (0,0)$$

*Sin embargo, como se discutió en la subsección 4.3.3, nuestro método realmente no comprueba este extremo. Es decir, conservativamente prohibe un tamaño de bloque si el I-test devuelve "yes" para las dos ecuaciones ($eq1$) y ($eq2$) correspondientes a ese tamaño de bloque.*

### 4.3.4. Búsqueda automática del tamaño de bloque

En aras de la productividad, es deseable que el usuario de la plantilla de wavefront no tenga que preocuparse por los detalles de implementación de un código wavefront con tiling. Entre estos detalles se encuentra la elección del tamaño del tile. En esta sección se explica una estrategia que permite realiza la búsqueda automática del bloque óptimo o cuasi-óptimo sin intervención del usuario.

La búsqueda del tamaño de bloque consta de tres pasos que enumeramos a continuación y que describimos con más detalle en los tres siguientes apartados:

1. Delimitación del espacio de búsqueda. En otras palabras, estimación del tamaño máximo de bloque recomendable.
2. Estudio de los pares $bi$, $bj$, dentro del rango calculado en el paso anterior, que no provocan ciclo de dependencias.
3. Búsqueda ortogonal del valor del tamaño de bloque recomendado.

#### 4.3.4.1. Delimitación del espacio de búsqueda

Uno de los inconvenientes del tiling es que reduce el paralelismo efectivo del problema. Como ya se ha comentado, con tiling las unidades de datos independientes están representadas por cada uno de los bloques o tiles. Cada tarea, ejecuta ahora un tile en lugar de una celda de la matriz. Por tanto, si el tamaño del tile es muy grande, el número de tiles puede ser muy pequeño y que se alcance un punto en el que no hay suficientes tiles independientes para mantener a todos los cores ocupados. Para computar el tamaño máximo de bloque nos basaremos en dos heurísticos que hemos validado experimentalmente:

1. Para mantener $n$ cores con un valor cercano al 100 % de ocupación, es necesario que al menos tengamos $1,5 \times n$ tareas independientes listas para ejecutar.
2. En los problemas en los que existe un transitorio (al principio o al final del problema) en el que el número de tareas independientes no es suficientemente grande

como para que todos los cores estén ocupados, el número de bloques que se recorren durante el transitorio no debe ser superior al 1 % del número total de bloques.

Más precisamente, aunque en principio podría parecer que con $nCores$ tareas independientes podemos mantener todos los cores ocupados, en nuestros resultados experimentales hemos comprobado que dichos cores no se ocupan al 100 % de su capacidad máxima. Esto es así por los retardos de sincronización debido la actualización de los contadores y un peor aprovechamiento de la cache por falta de libertad para reciclar las tareas en otras de la misma fila. Por tanto, para nuestra arquitectura de 32 cores necesitaremos problemas en los que cuanto antes tengamos $1,5 \times 32 = 48$ tareas independientes (es decir, bloques cuando implementamos tiling).

En segundo lugar, aunque hay problemas en los que desde el principio están listas para despachar varias tareas (como Checkerboard, Financial y Floyd en los que la primera fila del espacio de tareas está inicializada con contadores a cero), en otros problemas (como Smith-Waterman y H264), el problema lo dispara una única tarea. En este último caso, y dependiendo de la inclinación que presente el frente de onda, el número de tareas independientes va creciendo hasta alcanzar un máximo, y posteriormente el proceso se revierte y el número de tareas independientes se reduce hasta alcanzar de nuevo la unidad en el extremo opuesto de la matriz. Por ejemplo, en el problema Smith-Waterman, el máximo se alcanza en la antidiagonal de la matriz justo en la mitad de la ejecución, pero a partir de ese momento las sucesivas antidiagonales contienen menos bloques, hasta que el problema se vuelve a secuencializar al final del procesado. Con anterioridad, nos hemos referido con el término "régimen permanente" al periodo en que hay suficientes tareas independientes como para aprovechar todos los recursos hardware. Desde ahora, este término se refiere al tiempo en que hay $1,5 \times nCores$ tareas independientes. De forma similar, para estos problemas consideramos la existencia de un régimen transitorio de inicialización y otro de finalización de la computación. En media, durante los periodos transitorios los cores se ocupan mas o menos al 50 %. Por tanto, para mejorar la eficiencia la fracción de tiempo consumida durante los transitorios no debe suponer una fracción significativa del tiempo total. En nuestro heurístico, en vez de usar el tiempo como criterio, hemos visto más práctico usar el número de bloques que se procesan durante los transitorios. De esta forma, y basándonos en un abanico de pruebas experimentales, para este tipo de problemas proponemos que el número de bloques correspondientes al periodo transitorio no sea superior al 1 % del número de bloques totales.

Por ejemplo, para el problema Smith-Waterman con una matriz $40.000 \times 40.000$ y una arquitectura de 32 cores, se necesitan, como hemos calculado antes, 48 tareas independientes. Es decir, en una matriz de sólo $47 \times 47$ bloques nunca se alcanza el régimen permanente, ya que en el momento de máximo paralelismo sólo hay 47 tareas independientes. En una matriz de más bloques, el transitorio de inicio o arranque consume

$48 \times 47/2$ bloques y los mismos durante el transitorio de finalización. De esta forma, para que el 1 % del número total de bloques sea superior a $48 \times 47$ se debe cumplir que $bi \times bj < 0,01 \times 40.000^2/(48 \times 47)$. Es decir $bi \times bj < 7092$. Si consideramos un bloque cuadrado, $bi = bj < 84$ y que las dimensiones van a ser potencias de dos, un posible tamaño de bloque es $64 \times 64$. A partir de ahí, podemos encontrar otros tamaños válidos como $128 \times 32$, $32 \times 128$ o menores. En general, para una matriz de $m \times n$ y en una arquitectura de $nCores$, se debe cumplir que:

$$bi \times bj < \frac{0,01 \times m \times n}{1,5 \cdot nCores \times (1,5 \cdot nCores - 1)} \tag{4.10}$$

Si simplificamos para una situación en que $bi = bj$, y decimos que el tamaño máximo de bloque es *maxB*×*maxB*, podremos calcular *maxB* haciendo la raiz cuadrada de la expresión anterior.

Por último, para problemas como Checkerboard, Financial y Floyd, se buscará que el número de bloques independientes al inicio del programa sea mayor que $1,5 \cdot nCores$. Como el número de celdas independientes al inicio del programa es aproximadamente el tamaño de una fila, para una matriz de $m \times n$, la ecuación quedará como

$$maxB = bi = bj < \frac{n}{1,5 \cdot nCores}$$

#### 4.3.4.2. Validación de tamaños que no provocan ciclos

En la implementación de la plantilla que busca automáticamente el tamaño de bloque ideal para la ejecución más eficiente de un determinado código wavefront, debemos garantizar que los tamaños de bloque usados no provocan ciclo. Hemos decidido reducir el espacio de búsqueda a tamaños de bloque en el que $bi$ y $bj$ son potencia de dos, por tanto habrá que comprobar que los bloques entre $1 \times 1$ y *maxB*×*maxB* que cumplan esa condición no provocan ciclos de dependencias.

Durante esta comprobación podemos ahorrar algunos chequeos. Esto es así porque si para un $bi_1$, el I-test no encuentra solución entera para la $(eq1)$ de la ecuación 4.5, y se comprueba que no cambian los vectores de dependencias para otro $bi_2 > bi_1$, podemos estar seguros de que tampoco hay solución entera, y por tanto tampoco ciclo, para este nuevo tamaño de bloque. La razón se explica a continuación. Si la dimensión de la matriz es de $m \times n$, realmente tenemos $\lceil m/bi \rceil$ bloques en la dimensión $i$, por lo que ese será el número máximo de veces que podemos aplicar la componente $dx$ de los vectores de dependencia para la $(eq1)$. Así, realmente $N_k = \lceil m/bi \rceil$. De forma similar, para la $(eq2)$ tendremos $N_k = \lceil n/bj \rceil$. De cualquier modo, si no encontramos solución con un

valor alto de $N_k$, podemos estar seguro de que no la vamos a encontrar tampoco con un $N_k$ más pequeño. Es decir, si no encontramos una solución entera cuando las variables pueden moverse en el rango $[M_k, N_k]$, está claro que tampoco las vamos a encontrar si ahora el rango en el que se puede mover la variable es menor, $[M_k, N_k/b]$. De esta forma nos evitamos llamadas innecesarias a la función I-test.

En el código de la figura 4.15 describimos el algoritmo que devuelve una lista con los tamaños de tile que no provocan ciclo. La función `Valid_Tile_Sizes()` se apoya en la función `SameDep(bi1,bj1,bi2,bj2)` que devuelve `true` si las dependencias resultantes para el tamaño de bloque $bi1 \times bj1$ son las mismas que para el tamaño de bloque $bi2 \times bj2$. También nos apoyamos en la función `I-testEQ1(bi, dep)` que primero calcula los nuevos vectores de dependencias y la región en la que se aplican para un tamaño de bloque $bi$, y luego aplica el I-test a la primera dimensión, $(eq1)$, con los límites calculados. De forma similar, para $(eq2)$ hacemos uso de `I-testEQ2(bj, dep)`. Estas dos funciones devuelven `true` si se detecta un ciclo potencial, o `false` si podemos asegurar que no hay ciclo de dependencia para el tamaño de bloque especificado.

```
1 list Valid_Tile_Sizes(maxB){
2    bool last[maxB][maxB]=new bool[maxB][maxB];
3    for (int bi = 1; bi<maxB; bi*2){     //iteramos sobre bi
4       for (int bj = 1; bj<maxB; bj*2){  // iteramos sobre bj
5          bool cycle = false;
6          if (bi==1 || ! SameDep(bi,bj,bi/2,bj) || last[bi/2][bj])
7             if ( I-testEQ1(bi, dep) )
8                if ( bj==1 || ! SameDep(bi,bj,bi,bj/2) || last[bi][bj/2])
9                   if ( I-testEQ2(bj, dep) )
10                     cycle=true;
11         last[bi][bj] = cycle;
12         // no existe ciclo, incluimos el bloque en el conjunto.
13         if (!cycle) list_tile_sizes->add(bi,bj);
14      }
15   }
16 }
17 return list_tile_sizes;
```

Figura 4.15: Pseudocódigo para la búsqueda de ciclos.

Al principio del código creamos una matriz que nos permita recordar si tamaños de bloque ya procesados no provocan ciclo. A continuación dos bucles anidados recorren los valores de $bi$ y $bj$ potencia de dos entre 1 y *maxB*. Para el tamaño de bloque sin tiling, $1 \times 1$, se evalúa el I-test para las dos ecuaciones $(eq1)$ y $(eq2)$ de forma que comprobamos si el usuario ha especificado por error un patrón de dependencias potencialmente con ciclos. Dado que el I-test es conservativo, si el usuario está seguro de que su patrón

está libre de ciclos, existe un flag en la plantilla que fuerza el tamaño de bloque $1 \times 1$ como seguro. Para tamaños de $bi$, $bj$ mayores que 1, nótese que el algoritmo sólo llama al I-test para la ecuación correspondiente a la dimensión chequeada si las dependencias han cambiado con respecto a un tamaño de bloque inferior, o si no han cambiado, pero sí había ciclo para el tamaño anterior (líneas 8 y 6 del código). Si las dos llamadas al I-test devuelve true, `cycle=true` y se apunta dicha condición en la posición correspondiente de la matriz `last` (línea 11). Por último en la línea 13, añadimos a la lista `list_tile_sizes`, que contiene los tamaños de bloques válidos, el tamaño de bloque estudiado si es que no provoca ciclo. Esta lista es la salida de la función.

Un detalle de implementación no considerado en el código por no complicarlo más de lo necesario es que la matriz `last` está declarada con un tamaño mas grande de lo que necesario (*maxB*×*maxB*). Esto es así porque al recorrer $bi$ y $bj$, sólo interesan los valores potencia de dos, y por tanto la mayoría de las posiciones de `last` nunca se usan. Para corregirlo bastaría con que las dimensiones fuesen de $lg_2(maxB) \times lg_2(maxB)$ pero luego habría que indexar `last` de la forma `last[lg2(bi),lg2(bj)]`.

De cualquier modo, esta función se llama en tiempo de compilación de la plantilla y además el tiempo que consume es despreciable. Por ejemplo, para Smith-Waterman, con *maxB*=1024, la lista de bloques válidos se obtiene en 0,075 ms. En este caso, no hay ciclos para el tamaño $1 \times 1$ y las dependencias nunca cambian independientemente del tamaño de bloque de forma que sólo se llama a la función `I-test()` dos veces. De todas formas, una versión más simple del algoritmo mostrado en la figura 4.15 que ciegamente llama al I-test para cada par $bi$, $bj$ (100 llamadas en total), consume 0,2 ms. que sigue siendo inapreciable para el usuario que hace la llamada al generador de código a partir de la plantilla. Como ejemplo de un caso más caro tenemos el problema Checkerboard que si tiene ciclos cuando $bi > 1$. En este caso, nuestro algoritmo consume 0,48 ms. (o 0,51 ms. si usamos la versión simple). En este caso no hay tanta diferencia entre la versión optimizada y la simple ya que el ahorro sólo se produce para los tamaños $1 \times bj$ en los que no cambian las dependencias y la primera llamada al I-test devuelve "no". Sin embargo, para todos los demás $bi \times bj$ con $bi > 1$ el test devuelve "yes" y se siguen haciendo las llamadas por si un tamaño suficientemente grande dentro del rango deja de presentar ciclo.

#### 4.3.4.3. Búsqueda del tamaño recomendado

En este ultimo paso se busca de entre los tamaños de bloque, que no provocan ciclo, uno que resulte en un tiempo de ejecución cuasi-óptimo. Para ello, la idea es implementar un algoritmo rápido de búsqueda que tome muestras de los tiempos de ejecución para distintos tamaños de bloque y seleccione aquél que de lugar a un tiempo de ejecución más pequeño que con otros tamaños de bloque. Esta última frase hace referencia a

dos aspectos, primero, que se tomarán muestras de tiempos y segundo que se hará una búsqueda en la que nos conformamos con encontrar un mínimo local.

En cuanto al primer aspecto, evidentemente, para encontrar el mejor tamaño de bloque no podemos permitirnos ejecutar todo el código para una matriz de gran tamaño y todos los posibles tamaños de bloque. Pero tampoco es factible, para cualquier problema encontrar un modelo analítico que te permita estimar el tamaño de bloque adecuado. Por tanto, basaremos nuestra búsqueda en ejecuciones parciales del problema. En particular, una aproximación que nos ha dado resultados aceptables consiste en ejecutar el código con aproximadamente el 1 % de las celdas de la matriz. Más precisamente, arrancaremos la ejecución wavefront del código para los bloques centrales de la matriz. Para ello, los contadores deben ser inicializados de forma que simulen que todas las dependencias previas están resueltas y que se ha alcanzado (en tiempo cero) el régimen permanente del problema aproximadamente a la mitad del espacio computacional. El ejecutable que genera la plantilla ejecutará el problema de modo que se siguen cumpliendo las dependencias originales y el mecanismo de reciclado y gestión de contadores es similar al que sucede cuando ejecutamos el problema completo. Ya que queremos comparar el tiempo de ejecución para distintos tamaños de bloque, configuraremos cada problema para que se procesen el mismo número de elementos independientemente del tamaño de bloque. Por ejemplo, para un problema de tipo Smith-Waterman como el de la figura 4.16, temporizaremos la ejecución de la diagonal mostrada, que en este caso tiene 4 bloques de $2 \times 2$. Los contadores atómicos para estos cuatro bloques se inicializan a cero para simular que se ha alcanzado ese punto de la ejecución del problema.

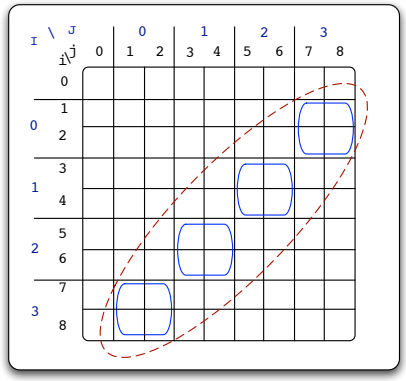
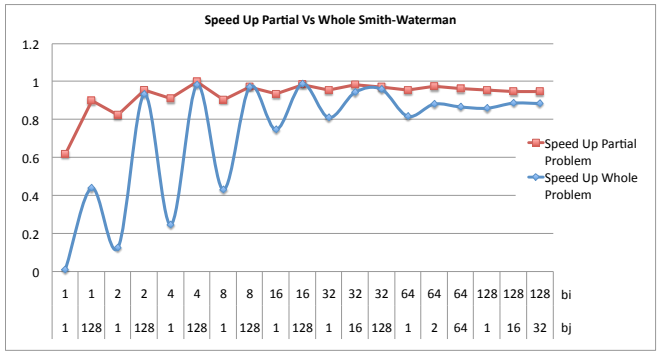
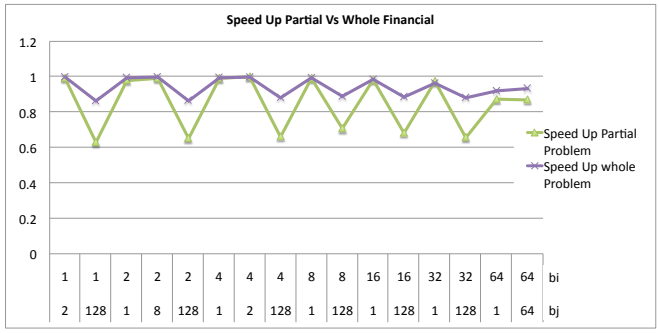


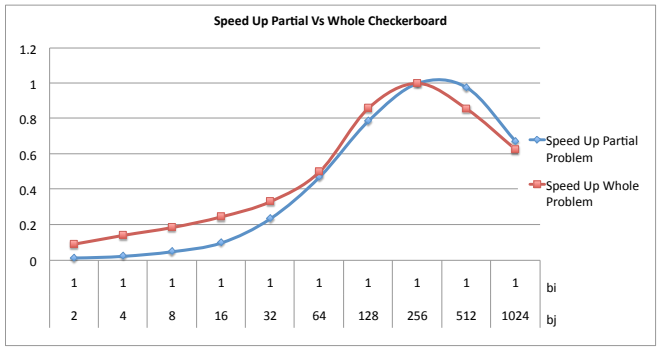

Figura 4.16: Selección de los elementos que se ejecutarán para tomar una muestra significativa del tiempo consumido durante la ejecución parcial de un problema tipo Smith-Waterman y un tamaño de bloque de $2 \times 2$.




(a)




(b)




(c)

Figura 4.17: Speedup normalizado respecto del mayor speedup conseguido para 32 cores ejecutando los problemas (a) Smith-Waterman, (b) Financial, y (c) Checkerboard, tanto la versión completa como la que toma una muestra de la ejecución del 1 % de la matriz. En el eje de abscisas representamos los valores de $bi$ en la línea superior, y de $bj$ en la inferior.

Para poder aplicar este algoritmo, debemos confirmar que el comportamiento de la ejecución parcial del problema es representativa de la ejecución completa. En la figura 4.17 mostramos para 32 cores y distintos tamaños de $bi \times bj$ el speedup normalizado (dividido entre el mayor valor de speedup) para cada problema completo así como para la ejecución de una muestra que sólo considera aproximadamente el 1 % de los elementos. En los tres casos vemos como las dos ejecuciones presentan los máximos y los mínimos para los mismos tamaños de bloque. Especialmente para el caso del problema Smith-Waterman, vemos como para la ejecución completa, las diferencias de speedup entre distintos tamaños de bloque son más evidentes, pero incluso en ese caso, la ejecución del problema parcial nos permite también encontrar un buen tamaño de bloque. Recordemos que para el problema Checkerboard los tamaños de bloque con $bi > 1$ provocan un ciclo de dependencias por lo que en la figura sólo se muestran tamaños de bloque del tipo $1 \times bj$.

Aunque en estas figuras no se reflejen todos los tamaños de bloques medidos, para estos tres problemas hemos comprobado como para cada tamaño de $bi$, hay un tamaño de $bj$ que hace que el rendimiento sea óptimo. Por ejemplo, para los problemas Smith-Waterman y Financial, podemos observar que, para todos los $bi$, el tiempo de ejecución va disminuyendo según vamos aumentando $bj$ hasta que llegamos al punto en el que el grado de paralelismo disponible empieza a disminuir demasiado. En ese punto las ventajas del tiling no compensan la falta de tareas independientes que permitan trabajar a todos los cores la mayor parte del tiempo.

Teniendo en cuenta este comportamiento observado, pasamos al segundo aspecto de esta búsqueda automática del tamaño de bloque. Explorar todos los posibles tamaños de bloque supondría tomar muestras de los tiempos de ejecución para los, potencialmente, $(lg_2(maxB))^2$ tamaños de bloque posibles. Por ejemplo, para *maxB*=256, el máximo número de tamaños de bloque, si ninguno provoca ciclos, es de 64. Pero el coste de tomar muestras para todos esos tamaños de bloques no es sólo muy elevado sino además innecesario. Hacemos esta afirmación ya que para los problemas estudiados, como hemos podido ver en la figura 4.17, aunque las diferencias de speedup entre los tamaños de bloque "buenos" y los "malos" puede ser bastante grande, dentro de los que consideramos "buenos" las diferencias son muy pequeñas. Es decir, encontrar un tamaño de bloque que proporcione un mejor speedup que los tamaños de bloque de dimensiones cercanas será considerado como suficiente. En otras palabras, daremos por aceptable encontrar un máximo local del speedup ya que todos los máximos locales se parecen tanto entre sí que no merece la pena invertir más tiempo en la búsqueda del máximo global.

Con estas consideraciones, y suponiendo un espacio de búsqueda bi-dimensional, diseñamos e implementamos un algoritmo basado en una búsqueda ortogonal. Este tipo de algoritmos se usan en varios ámbitos, como por ejemplo la fase de "block matching" de un algoritmo de compresión de video basado en compensación del movimiento. Como

ilustramos en la figura 4.18, la idea es empezar fijando una de las variables del espacio de búsqueda, en la figura $bi = 128$. Para ese $bi$, partiendo del mayor tamaño de $bj$, vamos tomando una muestra del tiempo de ejecución de cada $bi \times bj$ mientras el tiempo vaya disminuyendo (paso 1 en la figura). Cuando en algún paso el tiempo no disminuye, nos quedamos con el $bj$ anterior (el mínimo) y nos movemos ahora decrementado $bi$ en tanto en cuanto el tiempo disminuya (paso 2). Adicionalmente, por si al cambiar $bi$ existen valores cercanos de mejores $bj$, también tomamos muestras modificando $bj$ en los dos sentidos (pasos 3 y 4). Nótese que dado que los tamaños de bloque de gran tamaño suelen resultar los más acertados, empezamos por dichos valores. En particular, al aplicar la ecuación 4.10 a este problema con 32 cores, para $bi = 128$, el $bj$ máximo resultante es 32, y por tanto ese es el punto de partida de la búsqueda ilustrada en la figura 4.18. Claramente, este proceso no garantiza que encontremos el máximo global, pero como hemos argumentado en el párrafo anterior, un máximo local es suficiente para nuestro problema. Por ejemplo, en este caso hemos encontrado el bloque recomendado de $32 \times 8$ que, como veremos en los resultados experimentales, es sólo 6 centésimas de segundo más lento que el máximo global que está en $16 \times 64$. Además, por un lado, en el Checkerboard, este algoritmo encuentra el máximo global, y por otro el algoritmo de búsqueda es fácilmente modificable para que, en caso deseado, tome la muestra de los tiempos de ejecución para todos los tamaños de bloque posibles y elija el mejor. Esta búsqueda global podría tener sentido cuando el tiempo invertido en la búsqueda del mejor tamaño de bloque no es una limitación.

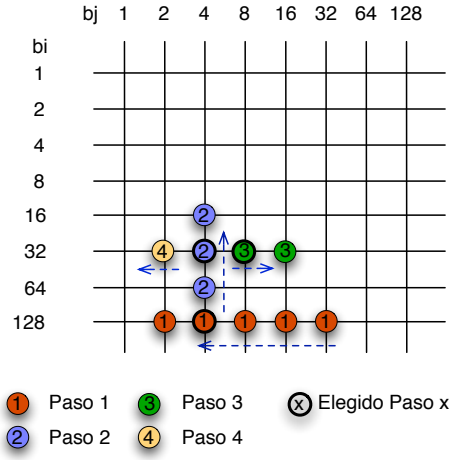



Figura 4.18: Ejemplo del proceso de búsqueda ortogonal.

El pseudo-código del algoritmo se muestra en la figura 4.19.

```
1 time bestTime = inf;
2 int bestI,bestJ;
3
4 bool lookForTheBest(bi,bj){
5    bool newIsWorst = false;
6    t1 = time();{
7    wave->search(bi,bj);
8    t2 = time();{
9    time = t2-t1;
10   if (time<bestTime){
11      bestTime = time;
12      bestI = bi; bestJ = bj;
13   }else
14      newIsWorst = true;
15   return newIsWorst;
16 }
17
18 pair<int,int> search(){
19   pair_bi_bj = tile_valid_size.find_max();
20   biM=pair_bi_bj.first; bjM=pair_bi_bj.second;
21
22   for(bj = bjM; bj >= 1 && !newIsWorst; bj /= 2 )
23      if( tile_valid_size.find(pair(biM,bj) )
24         newIsWorst = lookForTheBest(biM,bj);
25
26   for(bi = biM/2; bi >=1 && !newIsWorst; bi /= 2)
27      if( tile_valid_size.find(pair(bi,bestJ) )
28         newIsWorst = lookForTheBest(bi,bestJ);
29
30   oldBestJ=bestJ;
31   for(bj = bestJ*2; bj<=maxBj && !newIsWorst; bj *= 2)
32      if( tile_valid_size.find(pair(bestI,bj) )
33         newIsWorst = lookForTheBest(bestI,bj);
34
35   if(bestJ==oldBestJ)
36      for(bj = bj/2; bj>=1 && !newIsWorst; bj /= 2)
37         if( tile_valid_size.find(pair(bestI,bj) )
38            newIsWorst = lookForTheBest(bestI,bj);
39
40   write(bestI,bestJ); // Escribe en el fichero blocksize.wavefront
41   return make_pair(bi,bj);
42 }
```

Figura 4.19: Pseudocódigo del algoritmo para encontrar el mejor tamaño de bloque.

Entre las líneas 18 y 42 está definida la función `search`, que es la encargada de realizar la búsqueda del tamaño de bloque que tiene rendimiento óptimo. Esta función está formada por cuatro bucles `for` consecutivos con la misma estructura. El primero

(línea 22) efectúa la búsqueda del mejor tamaño de columna ($bestJ$) para el tamaño de fila $bi = 1$. El segundo `for` (línea 26) busca para el mejor tamaño de columna encontrado en el bucle anterior, el mejor tamaño de fila ($bestI$). El tercer (línea 31) y cuarto bucle (línea 36) buscan alrededor del tamaño de fila encontrado si existe algún mínimo que mejore la solución encontrada. Para ello, en el tercer bucle iteramos sobre el tamaño de columna buscando si hay algún tamaño de columna menor con mejor rendimiento que el seleccionado, mientras que en el cuarto buscamos si existe algún tamaño de columna mayor con mejor rendimiento. Cada uno de los bucles en el interior, primero comprueba si el tamaño de fila y tamaño de columna seleccionado están entre los bloques válidos, es decir, no es un tamaño de bloque provoque ciclo en las líneas 23, 27, 32 y 37 respectivamente. Esto se realiza invocando al método `tile_valid_size.find` para asegurar si está el tamaño de bloque $bi$, $bj$ en el conjunto. El método `find` de la clase `set` devuelve un puntero al elemento nulo en caso de no estar dicho elemento en el conjunto. Cada elemento del conjunto está formado por una pareja de enteros. Es por ello que utilizamos `pair` para convertir la pareja formada por el tamaño de fila y el tamaño de columna en un elemento para poder buscarlo en el conjunto de bloques válidos. Una vez comprobado si el tamaño de bloque es válido, llamamos a la función `lookForTheBest` (en las líneas 24, 28, 33 y 38) para verificar si el tamaño de bloque produce mejor rendimiento parcial hasta el momento. Esta función devuelve `true` en caso de que el tamaño de bloque no obtenga el mejor resultado parcial, condición de salida en todos los bucles. La función `lookForTheBest`, definida en la línea 4 de la figura, recibe como parámetros el tamaño de bloque para ambas dimensiones. Para este tamaño de bloque, llama a la ejecución parcial del problema (línea 7) midiendo el tiempo que tarda (líneas 6 y 8). En caso de que sea la ejecución más rápida hasta el momento, almacena el tamaño de bloque actual y el tiempo empleado como el mejor en las variables $bestI$, $bestJ$ y $besTime$ (entre las líneas 10 y 12). La función devuelve además un valor booleano con el valor `true` si este tamaño de bloque no es mejor que el previamente almacenado o `false` en caso contrario. Finalmente, en la línea 40 el programa escribe en un fichero el mejor tamaño de bloque encontrado para que pueda ser leído posteriormente.

Este proceso de búsqueda es opcional, ya que si el usuario de la plantilla wavefront ya conoce el tamaño de bloque óptimo sólo tiene que indicarlo. En la sección 4.5 se explica cómo se especifica el tamaño de bloque o como opcionalmente se invoca la búsqueda automática que acabamos de describir. Un poco más adelante, en la sección 4.6, discutimos los resultados experimentales que confirman las ventajas de hacer esta búsqueda automática del tamaño de bloque. Por otro lado, estudiando los resultados experimentales que explicamos en la sección 4.6.1, observamos que para el problema Smith-Waterman y el problema Checkerboard tiende a obtenerse mejor rendimiento cuanto mayor es el tamaño de columna. Además según los experimentos realizados en 4.6.3 vemos que la

variación del tamaño de fila es más significativa debido a la precarga de líneas de memoria como explicamos en dicha sección. Esto nos motiva a sugerir una modificación sencilla en el algoritmo. En este caso, fijaríamos el tamaño máximo de columna y fila e iríamos buscando la mejor fila para dicho tamaño de columna. Una vez encontrado el mejor tamaño de fila para esa columna, buscaríamos el mejor tamaño de columna para la fila encontrada. Por último buscaríamos alrededor de este elemento en su tamaño de fila.

La diferencia del algoritmo es fácil de implementar y en nuestros experimentos reducimos el número de ejecuciones del problema parcial en Smith-Waterman de 11 a 4, en Checkerboard de 9 a 3 mientras que en Financial aumentaría de 9 a 11. Como vemos, el ahorro de tiempo depende del problema en cuestión por lo que esta modificación en el algoritmo podría considerarse configurable por el usuario según las características del problema.

## 4.4. Simplificación de dependencias

El número de dependencias influye no sólo en el overhead introducido por la inicialización de los contadores, sino también en el tiempo que cada tarea dedica a actualizar los contadores de las tareas dependientes. En esta sección, proponemos reducir el número de dependencias. La idea es substituir el patrón de dependencias original por otro equivalente con menor número de vectores de dependencias. Este patrón debe respetar la ordenación de tareas que impone el patrón original, pero no tiene porqué conducir a una ordenación exactamente igual a la original. En otras palabras, buscamos una ordenación de tareas que por un lado "contenga" a la ordenación original pero usando menos vectores.

La simplificación de dependencias se basa en la transitividad de las dependencias: si C depende de B y B depende de A, C depende de A. Enunciando esta idea desde otro punto de vista, si C depende de A, podemos encontrar un B tal que C depende de B y B depende de A que nos permita expresar la dependencia $A \rightarrow C$ en términos de $A \rightarrow B \rightarrow C$. Volviendo al dominio de las dependencias espaciales en un problema wavefront, y por empezar con un ejemplo, en la figura 4.20(a) ilustramos el patrón de dependencias asociado al vector de dependencias `(1:4,2)`. Este vector representa cuatro dependencias y por tanto implica la actualización de cuatro contadores por cada celda no sólo durante la inicialización de la matriz de contadores sino también al actualizar los mismo durante la ejecución. Sin embargo es fácil ver que esas cuatro dependencias se pueden simplificar por sólo dos: `(1,2);(1,0)`. En la figura 4.20(b) vemos como la dependencia entre los puntos $A$ y $C$ se sigue respetando, aunque ahora a traves del punto $B$ que depende de $A$ (por `(1,0)`) y del cual depende $C$ (por `(1,2)`). De forma similar

pasa con los puntos $D$ y $E$ de la figura.

Desde el punto de vista del paralelismo efectivo, la opción de la figura 4.20(b) tiene una ordenación más restrictiva que la correspondiente de la figura 4.20(a). Esto es claro, ya que inicialmente, $C$ se puede ejecutar en cuanto $A$ termine, y tras la simplificación, $C$ no sólo espera a que termine $A$ sino también $B$, por tanto la ejecución de $C$ se ve retrasada. Sin embargo, a efectos prácticos, cuando el tamaño de la matriz es suficientemente grande, esta restricción no conlleva contraprestaciones en términos de rendimiento. Es decir, con unas dimensiones de la matriz suficientemente grandes en comparación con el número de procesadores, la simplificación de dependencias puede retrasar ligeramente el momento en el que todos los procesadores están ocupados (régimen permanente de la ejecución wavefront), pero una vez que se alcanza ese punto, dicha simplificación no implica ningún overhead adicional.

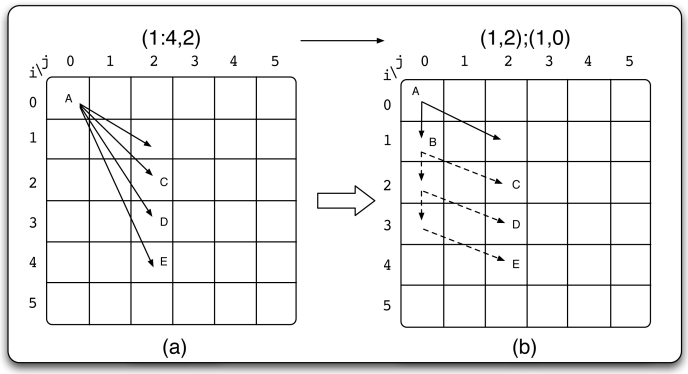



Figura 4.20: Ejemplo de simplificación de dependencias.

Además, no siempre existe este retraso ya que en varios problema de tipo wavefront el número de tareas que pueden arrancar al inicio del problema puede ser suficiente para ocupar todos los cores desde el principio. Por ejemplo, en los problemas Checkerboard, Financial y Floyd, la primera fila de la matriz de contadores está inicializada a cero, con lo cual todas las tareas correspondientes se despachan desde el principio.

Por último, cuando el ahorro en el número de dependencias es significativo y el grano de la tarea es fino, reducir el número de actualizaciones de contadores de dependencias compensa con creces el mínimo overhead que se podría introducir por añadir dependencias que no estaban en el problema original. Por ejemplo, el problema Financial presenta un patrón de dependencias descrito por `[1:m-1,1:n-1]->(1,0:n-j-1)`, patrón que representamos en la figura 4.21(a). Por tanto, para cada tarea tenemos `n-j-1` de-

pendencias (siendo `n` la segunda dimensión de la matriz, 4 en la figura, y `j` la columna de la tarea en cuestión). Este número de dependencias puede ser muy grande cuando trabajamos con matrices de grandes dimensiones, y sin embargo una simplificación de dependencias puede reducir el número a dos, en el caso general. El nuevo patrón trasformado se ilustra en la figura 4.21(b), donde vemos que sólo usando los dos vectores `(1,0)` y `(0,1)` se siguen respetando las dependencias originales y que este patrón contiene (o comprende) al patrón original. Por ejemplo, en el nuevo esquema, la celda (2,2) sigue teniendo que esperar a que termine la (1,1), pero ahora de forma indirecta, ya que depende de la (2,1), que a su vez dependen de la (1,1).

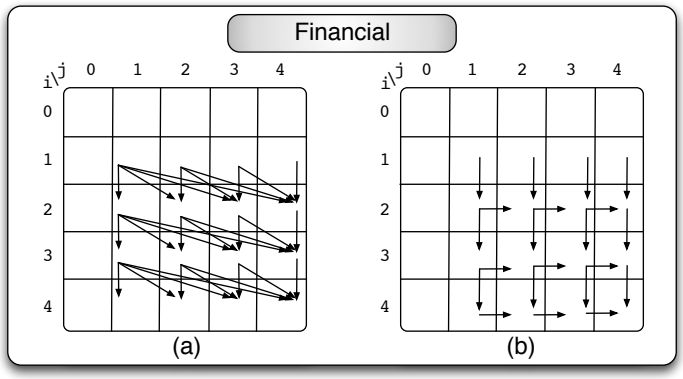



Figura 4.21: Proceso para simplificar las dependencias en Financial.

Volviendo a la cuestión del rendimiento, vimos, en la figura 4.1 de la sección 4.1 al principio de este capítulo, que el speedup que conseguíamos ejecutando el problema Financial implementado con nuestra plantilla original está cercano a 3.5 para 32 cores. Sin embargo, usando el vector de dependencias simplificado, `(1,0);(0,1)`, alcanzamos un speedup de 25. Si cuantificamos el número de actualizaciones de contadores atómicos que se hacía en el problema original, resulta en $(m-2)\cdot(n-1)\cdot n/2$, mientras que ahora se hacen $2(m-1)\cdot(n-2)+m-2$. Es decir, si la matriz es cuadrada, pasamos de $O(n^3)$ a $O(n^2)$. Además, teniendo en cuenta que los contadores son atómicos, se reduce significativamente la contención. Es decir, antes podía haber contadores que se decrementasen hasta $(m-1)$ veces (los contadores de la última fila), pero ahora sólo dos tareas, como mucho, contienden para decrementar un contador. De todas formas, estos aspectos, aún siendo importantes, no justifica en su totalidad el incremento en speedup. El factor que nos falta por tener en cuenta es la localidad. Primero, porque actualizar $O(n^2)$ contadores provoca menos fallos de caché que actualizar $O(n^3)$. Y segundo, porque al introducir la dependencia (0,1) como primera dependencia de cada

tarea, fomentamos que la tarea nueva en la que se recicle cada tarea sea la de la celda consecutiva en su misma fila, y por tanto, como explicamos en el capítulo anterior, se aproveche la localidad espacial, lo que redunda significativamente en el rendimiento. En otras palabras, conseguir una ordenación alternativa de tareas que, respetando el orden original, consiga explotar mejor la localidad y reducir la contención en el acceso a los contadores atómicos conduce a importantes mejoras en la eficiencia del código, aún a costa de reducir el grado de paralelismo del problema.

### 4.4.1. Proceso de simplificación de dependencias

Para simplificar las dependencias partimos de dos premisas:

- Recordamos que dado un vector de dependencias para una región, descrito por `[lx:hx, ly:hy]-> (dlx:dhx, dly:dhy)`, el vector se aplica a cada punto de la región `[lx:hx, ly:hy]`. Por razones prácticas, expresaremos el vector de dependencias con su forma más completa, incluyendo el "stride" o paso, es decir mediante la notación: `(dlx:dhx:sx, dly:dhy:sy)`. No conocemos problemas reales en los que el stride sea mayor que uno, así que por comodidad hemos omitido el término. De cualquier modo, si el stride es mayor que uno, cambiarlo a la unidad sigue respetando la ordenación original del problema wavefront. Por tanto, lo que en realidad nos interesa del stride es su signo.
- Todo vector de dependencias puede ser expresado como una combinación lineal de los vectores de la base ortonormal (vectores ortogonales de norma 1), usando para ello las coordenadas cartesianas del vector como coeficientes de la combinación lineal. Por ejemplo, el vector `(3,5)` no es más que `3(1,0)+5(0,1)`. Esto también es aplicable a vectores compuestos del tipo `(dlx:dhx, dly:dhy)`, tras su desenrollamiento en vectores simples. Por ejemplo, el vector `(1:2,5:6)`, representa a los vectores simples `(1,5),(1,6),(2,5),(2,6)`.

Además, desde el punto de vista de las dependencias, cualquier dependencia del tipo `(0,i), i>1`, puede substituirse por la dependencia más restrictiva `(0,1)` (de forma similar para cualquier dependencia de norma mayor que 1 que sea combinación lineal de un solo vector perteneciente a la base ortonormal). Por ejemplo, si en una región 2D sólo tenemos la dependencia `(3,0)`, esta puede substituirse conservativamente por la `(1,0)`, a costa de perder cierto grado de paralelismo: en el problema original hay bloques de tres filas consecutivas de la matriz independientes entre si, mientras que en el problema transformado cada fila depende de la anterior. Si el tamaño de una fila es suficientemente grande, el problema transformado también consigue ocupar todos los cores y no incurre en una penalización temporal.

Con todo esto podemos discutir tres posibles implementaciones para la simplificación de dependencia:

1. La solución más conservadora pasa por reemplazar el vector `(dlx:dhx:sx, dly:dhy:sy)` por los vectores de la base con el signo que corresponda a cada dimensión: `sx(1,0);sy(0,1)`. Por ejemplo, un vector `(1:3,-3:-5)` se substituiría por `(1,0);(0,-1)`.

2. Si el vector es del tipo `(dlx:dhx:sx, dy)`, ordenado de forma $|dlx| < |dhx|$, se puede transformar por `(dlx,dy);sx(1,0)`. Esta transformación se ha ilustrado en la figura 4.20. De forma similar, para los vectores `(dx,dly:dhy:sy)`, ordenado de forma $|dly| < |dhy|$, se puede transformar por `(dx,dly);sy(0,1)`. Esta transformación se ilustra en la figura 4.22, para el caso `(2,-1:-4)` que se simplificaría por `(2,-1);(0,-1)`. Esta alternativa reduce el grado de paralelismo en menor medida que la anterior.

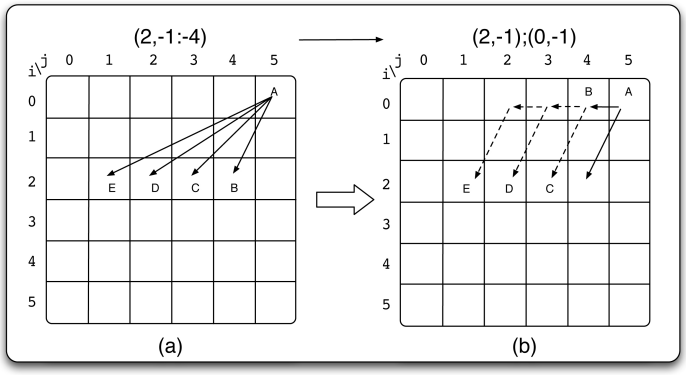



Figura 4.22: Ejemplo de simplificación de dependencias con `sy` negativo.

3. Y por último, una mejora de la anterior, que en lugar de crear dependencias con celdas que inicialmente no dependen de ninguna, desplaza dichas dependencias a la región en la que realmente son necesarias. Por ejemplo, el caso de la figura 4.20 quedaría ahora como en la figura 4.23, donde el vector `(1,0)` se aplica sólo en la región `[1:5,2:5]` (aunque el vector `(1,2)` se sigue aplicando en toda la matriz).

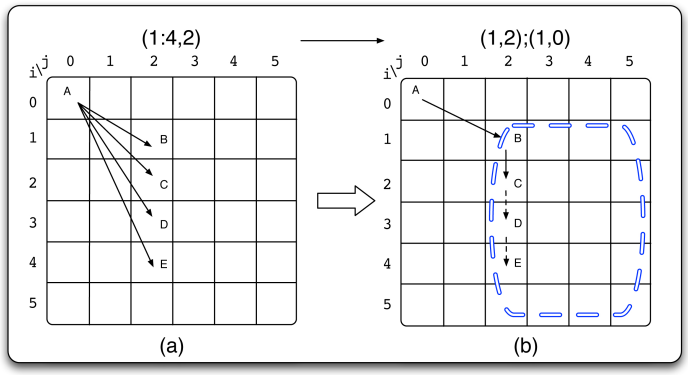



Figura 4.23: Proceso para simplificar las dependencias aplicando el vector de la base en una región de menor tamaño, equivalente al ejemplo de la figura 4.20.

De forma similar, el caso de la figura 4.22 quedaría ahora como en la figura 4.24, donde el vector `(0,-1)` se aplica sólo en la región `[2:5,0:4]`.

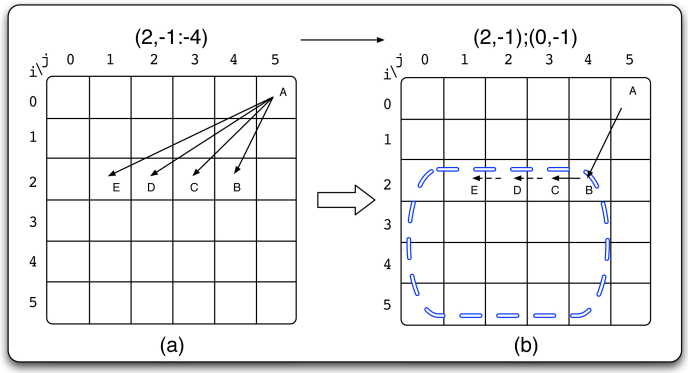



Figura 4.24: Proceso para simplificar las dependencias aplicando el vector de la base en una región de menor tamaño, equivalente al ejemplo de la figura 4.22.

Esta última opción reduce aún menos el grado de paralelismo disponible (número de tareas independientes) y es de fácil implementación ya que sólo hay que calcular la región en la que aplicar el vector de la base.

Tras evaluar experimentalmente, en nuestra arquitectura de 32 cores, las tres alternativas de simplificación de direcciones para matrices de gran tamaño (por otro lado nece-

sario para que tenga sentido ejecutar el código wavefront en paralelo), se comprobó que ninguna incurre en una reducción del paralelismo disponible como para afectar al tiempo total de ejecución. En los tres casos y para distintos problemas, se alcanza casi inmediatamente el régimen permanente en el que los 32 cores encuentran tareas independientes que procesar. Por tanto, en la práctica se decidió implementar en la plantilla la primera de las soluciones descritas, ya que es la mas sencilla y general.

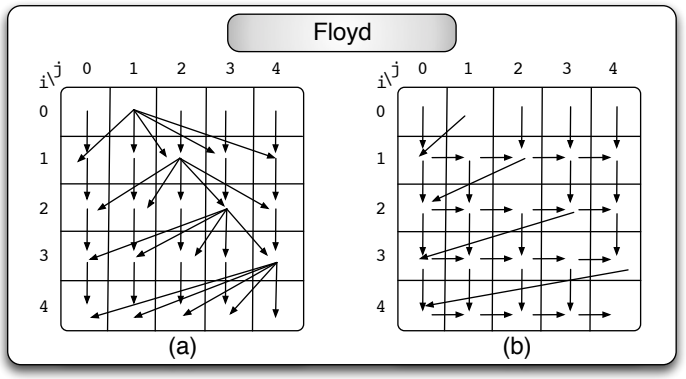



Figura 4.25: Proceso para simplificar las dependencias en el algoritmo de Floyd.

Sin embargo, no en todos los casos es recomendable aplicar esta simplificación de dependencias. Como ya hemos visto, la reordenación de tareas supone una perdida de paralelismo en los problemas. Sin embargo, habitualmente esta perdida de paralelismo no es suficientemente importante como para influir en el rendimiento. No obstante, en el algoritmo de Floyd tras aplicar la reducción de las dependencias se produce una serialización bastante grande del código. En la figura 4.25(a) mostramos el patrón de dependencia original del problema de Floyd y en la figura (b) la versión simplificada. En esta última sólo nos quedamos con las dependencias (0,1) y (1,0) para todos los elementos, mientras que para los elementos de la superdiagonal tenemos además la dependencia (1,-j). En el dibujo del patrón de dependencias simplificado podemos comprobar que existe mucha secuencialidad. En el primer instante se pueden ejecutar todas las tareas de la fila 0. Sin embargo, en el siguiente instante sólo se puede ejecutar la tarea (1,0). Un periodo más tarde se podrá ejecutar la tarea (1,1). Sin embargo, la tarea (2,0) depende de la tarea (1,2) por lo que no existirán más tareas en paralelo hasta que ésta se haya ejecutado. Como vemos, la fase inicial del wavefront en este caso parece ser bastante secuencial. Por otro lado, peor aun es la fase final: la última fila del wavefront se ejecutará de forma totalmente secuencial debido a que el primer elemento de la última fila depende del último elemento de la fila anterior, que además será el último en ejecutarse de la fila debido

a la dependencia (0,1) introducida. Es decir, las últimas filas dependen siempre de uno de los últimos elementos de la fila anterior por lo que se serializa su ejecución. Por este motivo, esta optimización es opcional. Para activarla, el usuario deberá indicar el flag `--simplify` a la hora de llamar al script que configura la plantilla (`wavefront.sh --simplify < def.file` ).

## 4.5. Plantilla wavefront optimizada

En este trabajo hemos proporcionado una plantilla de alto nivel para implementar problemas de tipo wavefront. Esta plantilla permite aplicar la técnica de tiling al problema y opcionalmente llevar a cabo una búsqueda automática de un tamaño de bloque recomendado (ver apartado 4.3.4). En esta sección describimos, desde el punto de vista del usuario, como sacar provecho de esta plantilla y daremos algunos ejemplos de los ficheros que se deben proporcionar para distintos problemas reales.

El usuario tan solo tiene que escribir un fichero de definición y una cabecera, el fichero `userMethods.h`, que incluye los dos siguientes métodos:

1. `dataInit(int argc, char ** argv)`: función encargada de inicializar los datos de la matriz. Tiene acceso a los argumentos del programa ya que los recibe de la función `main`.

2. `executeTask()`: método que especifica la computación necesaria para cada celda de la matriz. Esta es la función que llama cada tarea para una celda de la matriz o para varias si el tiling está en uso.

En la figura 4.26, mostramos un esquema general del proceso de creación del código ejecutable a partir de estos dos ficheros. En la parte de arriba aparecen los dos ficheros necesarios, el de definición que describe el patrón de dependencias y el fichero cabecera con los dos métodos descritos. A continuación el usuario debe llamar al script `wavefront.sh` pasándole como entrada el fichero de definición. Si en esta llamada se especifica el argumento opcional `--tile` se generará la versión wavefront que soporta cualquier tamaño de bloque $bi \times bj$. Si se especifica el argumento opcional `--simplify` se aplicará la optimización de simplificación de dependencias. El script `wavefront.sh` llama internamente al ejecutable que contiene todas las rutinas para generar los ficheros fuentes con la implementación wavefront del problema en cuestión. Como vemos en el esquema de la figura 4.26, estos ficheros fuentes generados por el script son los siguientes:

1. `main_wavefront.c++`: Este fichero contiene la rutina principal del programa que implementa el wavefront. Incluye los dos métodos codificados por el usuario en `userMethods.h` además del fichero cabecera `wavefront.h` del que hablamos a continuación.

2. `wavefront.h`: Contiene los métodos principales necesarios para la ejecución wavefront:`getDependency()`, `getCounter()` y `wavefront_init()`. Estos métodos se explicaron en el capítulo anterior.

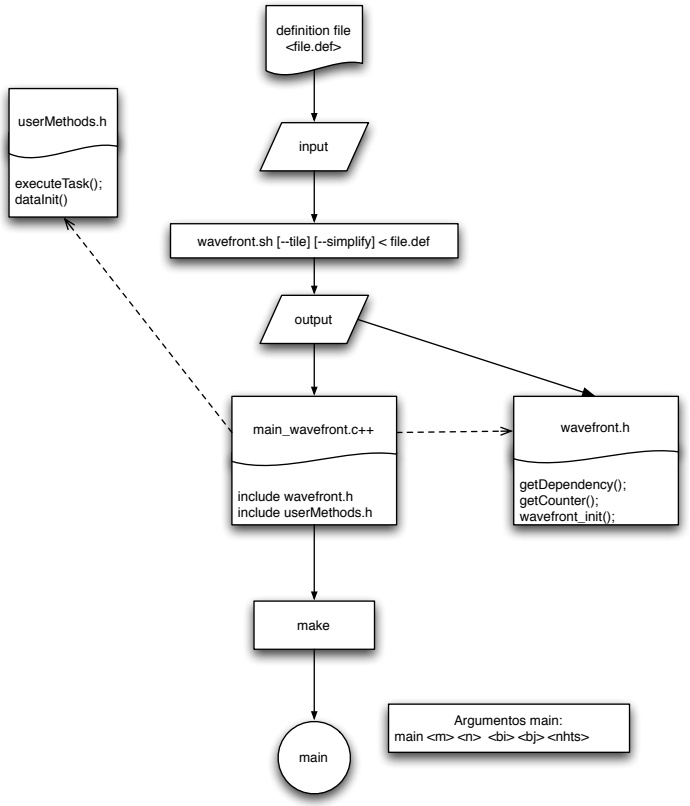



Figura 4.26: Esquema de ficheros necesarios y uso de la plantilla wavefront optimizada.

Con estos ficheros podemos invocar al comando "`make`" que llama al compilador de C++ y genera el fichero ejecutable `main`. Para los problemas 2D, en el ejecutable

`main` el convenio elegido ha sido reservar los primeros cinco parámetros de forma que invocamos al ejecutable usando el comando:

```
./main <m> <n> <bi> <bj> <nthreads>
```

siendo $m \times n$ el tamaño de la matriz de datos, $bi \times bj$ el tamaño de bloque y `nthreads` el número de threads con que queremos que se ejecute el código. También por convenio, si introducimos $bi$ y $bj$ mayores que cero, esos serán los valores para el tamaño de bloque, pero si especificamos un 0 para $bi$ y $bj$, estamos solicitando a la plantilla que use un valor recomendado automáticamente. En este caso, si es la primera vez que se llama al `main` con estos valores se realiza el proceso de búsqueda y se genera un archivo `blocksize.wavefront` con los tamaños de $bi$ y $bj$ recomendados, seguido de la ejecución del problema con ese tamaño de bloque. Si el fichero `blocksize.wavefront` ya existe, nos ahorramos la búsqueda y directamente pasamos a ejecutar el problema con el tamaño de bloque almacenado en dicho fichero.

```
1 #include "wavefront.h"
2 pair<int,int> getBestBlock (int argc, char **argv)
3 {
4   pair<int,int> block_size;
5   //Procesado de argumentos
6    ...
7    bi=atoi(argv[3]);
8    bj=atoi(argv[4]);
9    block_size=make_pair(bi,bj);
10   ...
11   if (bi==0 || bj ==0)
12      if ( readBlockSize(``blocksize.wavefront'', &block_size) == NULL)
13          block_size=search(); // Ver figura 4.19
14  return block_size;
15 }
16
17 int main (int argc, char **argv)
18 {
19   pair<int,int> blocks;
20   threads=processArguments(argc, argv);
21   ....
22   tbb::task_scheduler_init init(threads);
23   blocks=getBestBlock(argc, argv);
24   dataInit(argc, argv);
25   wavefront_init(blocks);
26   wavefront->run();
27 }
```

Figura 4.27: Resumen del contenido del fichero main_wavefront.c++.

Este proceso se ilustra en la figura 4.27 donde mostramos las líneas más representativas del fichero `main_wavefront.c++`. En la línea 20 de la figura se procesan los argumentos de entrada del problema y así obtenemos el número de threads. A continuación en la línea 23 realizamos una llamada a la función `getBestBlock`, la cual devuelve el tamaño de bloque recomendado. En caso de que el usuario haya indicado un tamaño de bloque $bi = 0$ y $bj = 0$, este método busca si existe el fichero `blocksize.wavefront` (línea 12). En caso de que exista, se obtiene de ahí el tamaño de bloque que se encontró en una búsqueda previa. Si el fichero no existe, se llama a la función `search` en la línea 13. Finalmente, en la línea 14 se realiza la devolución del tamaño de bloque recomendado, que se usa posteriormente para incializar el problema y lanzar la ejecución wavefront.

Otra posibilidad que puede ser implementada es la de que el usuario sólo quiera buscar el mejor tamaño para una de las dimensiones del bloque, $bi$ o $bj$. En ese caso, puede especificar con un número mayor que cero la dimensión que quiere fijar y dejar a cero la otra. El sistema respetará el tamaño de la dimensión fijada por el usuario y se limitará a buscar el tamaño para la otra dimensión que reporte un menor tiempo de ejecución. Por ejemplo, si se invoca al programa con $bi = 0$ y $bj = 64$, el ejecutable buscará sólo el $bi$ para el que se mida el mejor rendimiento.

Antes de terminar esta sección queremos recordar que los ficheros fuentes generados, `wavefront.h` y `wavefront.c++` son accesibles al programador y que por tanto, si se desea se pueden modificar u optimizar manualmente.

## 4.6. Resultados experimentales

En este apartado presentamos los resultados obtenidos con esta nueva plantilla optimizada para desarrollar códigos wavefront. Recordemos que las tres optimizaciones implementadas son, i) inicialización dinámica de contadores, ii) tiling y iii) simplificación de dependencias. La dos últimas optimizaciones se puede desactivar de forma independiente y desacoplarlas de la primera optimización. En la sección anterior se explica que al invocar al script `wavefront.h` se puede activar el tiling con `--tiling` y la simplificación de dependencias con `--simplify`. Sin embargo, dado que el estudio sobre el incremento de rendimiento mediante la primera optimización, discutido al final de la sección 4.2, demostró su eficacia, en la nueva plantilla esa técnica está siempre activa.

Los códigos elegidos para la evaluación experimental que presentamos a continuación son Smith-Waterman, Financial y Checkerboard. Estos son los tres problemas que en la sección 4.1 presentaron muy poca o ninguna escalabilidad debido a su baja gra-

nularidad. En cuanto al problema H264, éste ya trabaja a nivel de macrobloque, MB, o submatrices de la imagen de dimensión $16 \times 16$. Es decir, en cierto modo H264 ya implementa tiling, y lo hace con un tamaño de bloque adecuado. Ademas, si decidimos agrupar macrobloques, tampoco disponemos de mucha flexibilidad para elegir el tamaño de bloque ya que debido al patrón de dependencias del H264, sólo los tamaños del tipo $1 \times bj$ son válidos (otros con $bi > 1$ provocan ciclo). Por último, dado que para los vídeos de alta resolución las matrices son de $1280 \times 720$ los macrobloques de $16 \times 16$ ya están bien dimensionados y hacer tiles de macrobloques no reporta mayores beneficios, mas al contrario reduce el número de bloques independientes y el rendimiento empeora. Por su parte, el algoritmo de Floyd también ha sido descartado de este estudio ya que también presentaba una granularidad muy gruesa. Recordemos que ese problema realmente procesa un dominio de datos tridimensional y que para cada celda de la malla que procesamos en wavefront hay que calcular el mínimo de los valores en dicha tercera dimensión (un vector).

Resumiendo, hemos realizado cuatro tipos de experimentos enfocados a evaluar distintos aspectos relativos a la ejecución de los códigos wavefront generados por la nueva plantilla optimizada:

1. Estudio del impacto del tamaño de bloque en la aceleración de las aplicaciones.
2. Coste de la búsqueda ortogonal del tamaño de bloque recomendado.
3. Relación entre la aceleración de los códigos y la intensidad aritmética para cada tamaño de bloque.
4. Relación entre la aceleración y el consumo energético.

Abordamos estos puntos en las tres siguientes secciones. Los tres primeros estudios se ha realizado usando una arquitectura multicore con 32 procesadores Intel Xeon ® CPU X7550 a 2 Ghz. Los códigos fueron compilados con la versión 12 de icc y la opción de optimización -O3. Los resultados que se muestran son la media de los obtenidos en 10 ejecuciones consecutivas.

## 4.6.1. Impacto del tamaño de bloque en el rendimiento

#### 4.6.1.1. Smith-Waterman

Empezando con el problema Smith-Waterman, en la figura 4.28 presentamos el speedup obtenido para secuencias de 40.000 caracteres (matriz de 40.000$\times$ 40.000) con distintos tamaños de bloque y distinto número de cores. En esta gráfica no se han represen-

tado todos los tamaños de bloque, sino sólo los que presentan alguna singularidad, como dar lugar a un speedup máximo o mínimo para algún número de cores.

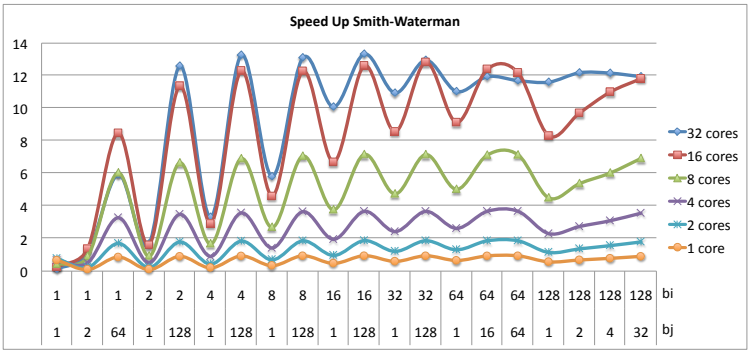



Figura 4.28: Speedup del problema Smith-Waterman para distintos tamaños de bloque $bi \times bj$ y distinto número de cores.

Vemos como para el tamaño $1 \times 1$, independientemente del número de cores, el speedup está en torno a 1. Esto coincide con las conclusiones ya discutidas en la sección 4.1 y los resultados de la figura 4.1. Resumiendo, sin las optimizaciones contempladas en la plantilla de wavefront optimizada, el problema Smith-Waterman tiene tareas con muy poca carga computacional y la ejecución paralela no compensa los overheads de la ejecución wavefront. También es apreciable que incluso con 1 core (ejecución secuencial) la elección del tamaño de bloque tiene bastante impacto. Por ejemplo, con tamaños de bloque $2 \times 1$ o $1 \times 2$ el speedup sobre el secuencial es 0,12 (es decir un 12 % de eficiencia) mientras que con un tamaño de bloque de $16 \times 128$ la eficiencia es del 98 %.

Para 2, 4, 8 e incluso 16 cores, el problema escala suficientemente bien si se configura un tamaño de bloque adecuado. En la figura 4.1 se ve que los mejores speedup se alcanzan para bloques de $4 \times 128$, $8 \times 128$, $16 \times 64$, $32 \times 128$, o incluso $64 \times 64$. Las diferencias entre estas speedup no son destacables. Por ejemplo, para 16 cores esos speedup máximos tienen una diferencia máxima entre ellos del 5 %, y ese rango de variación baja a 3,5 %, 2,8 % y 1,6 % para 8, 4, y 2 cores respectivamente. De esta información se desprende que los tamaños de bloque grande reportan reducciones en el tiempo de ejecución mientras no sean demasiado grandes como para que no haya suficiente paralelismo que explotar. Otra conclusión es que los tamaños de bloques "horizontales" (con más columnas que filas, $bj > bi$) consiguen mejores resultados que los tamaños cuadrados. Esto es así porque, sin llegar a reducir mucho el número de bloques independientes, permiten explotar mejor la localidad espacial ya que la ejecución dentro del bloque sigue un recorrido "row-wise" (por filas). Es decir, en este problema de Smith-Waterman en el

que apenas hay localidad temporal (cada celda se escribe una vez y se lee dos veces, una de ellas desde la fila inferior), primar los bloques horizontales que sacan mayor ventaja del recorrido por filas de la matriz es más razonable.

También vemos como para 32 cores el problema deja de escalar (el speedup apenas llega a 13 y la eficiencia baja del 40 %) y que para algunos tamaños de bloque se pueden obtener peores resultados que para 16 threads. Hay que subrayar que la arquitectura sobre la que se ejecutan estos experimentos tiene 4 sockets con un procesador de 8 cores cada uno. Cuando los 8 cores están ocupados tienen que compartir el bus de acceso a memoria y en problemas con tan poca localidad y tan baja carga computacional como Smith-Waterman es difícil que se mantenga la escalabilidad. Que el problema está limitado por el acceso a memoria se comprueba ejecutando el problema para tamaños más grandes. Por ejemplo, con matrices de $50.000 \times 50.000$ el speedup en 32 cores sube ligeramente a 13,6 y ya con matrices de $60.000 \times 60.000$ baja a 13,4. Es decir, aumentar el tamaño del problema no aumenta la relación entre el número de operaciones aritméticas respecto del número de "load" y "stores". En las siguientes secciones se aborda un estudio más detallado en el que se tiene en cuenta la intensidad aritmética y los fallos de caché en 32 cores.

Para un estudio más exhaustivo del impacto del tamaño de bloque en el speedup, la figura 4.29 muestra el speedup normalizado respecto del mayor speedup para todos los tamaños de bloque. En esa gráfica vemos como efectivamente, para valores de $bi$ iguales o inferiores a 16, aumentar el valor de $bj$ es siempre positivo y las curvas de speedup crecen cuando aumenta $bj$. Sin embargo, para $bi$ igual a 32, 64 o 128 empezamos a ver el efecto de la pérdida de paralelismo efectivo cuando $bi \times bj$ empieza a ser demasiado grande.

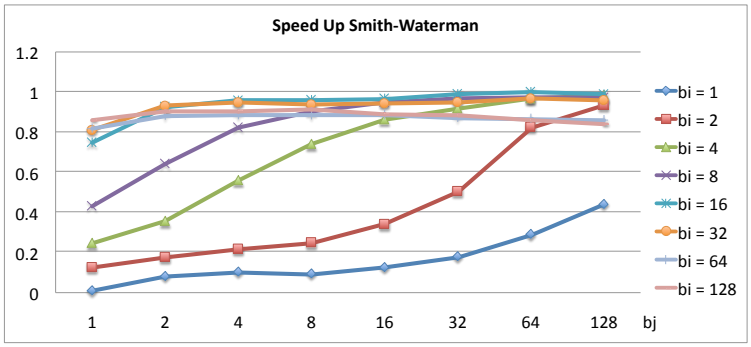



Figura 4.29: Speedup normalizado para el problema Smith-Waterman en 32 cores y todos los tamaños de bloque, $bi \times bj$, válidos.

#### 4.6.1.2. Financial

En la figura 4.30 mostramos el speedup alcanzado con distinto número de cores y distintos tamaños de bloque para el problema Financial. Configuramos el problema para el caso en que depositamos 10.000 euros en 10.000 bancos, es decir, para una matriz de $10.000 \times 10.000$.

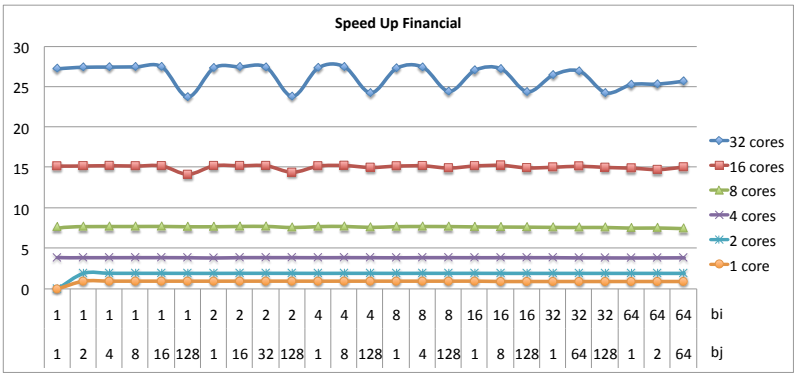



Figura 4.30: Speedup del problema Financial para distintos tamaños de bloque $bi \times bj$ y distinto número de cores.

Como se desprende rápidamente de la figura, estamos ante un problema con un comportamiento bastante diferente al del problema Smith-Waterman. Vemos que para 1 y 2 cores, en cuanto usamos la plantilla optimizada para tamaños de bloque mayores que $1 \times 1$ el speedup se estabiliza cerca de 1 y de 2 respectivamente, e independientemente del tamaño de bloque. Para 4 y 8 cores los códigos también exhiben una eficiencia cercana al 100 %. Con 16 cores la eficiencia apenas supera el 93 % y empieza a notarse una ligera pérdida de speedup para bloques muy horizontales de gran tamaño, $1 \times 128$ y $2 \times 128$. Para 32 cores la escalabilidad sigue siendo aceptable con un speedup máximo de 27,5 (eficiencia del 84 %) para el tamaño de bloque $1 \times 16$. Hay otros tamaños de bloque que reportan un speedup cercano a 27, pero también encontramos que los tamaños de bloque en que $bj = 128$ presenta speedup mínimos de un valor en torno a 24.

Para estudiar este fenómeno, en la figura 4.31, mostramos el speedup normalizado con respecto el speedup máximo para todas las combinaciones de $bi \times bj$ y ejecutando en 32 cores. Vemos como a partir de $bi \times 64$ el speedup empieza a decaer y para $bi \times 128$ se alcanza el mínimo. Estos tamaños de bloque tan horizontales reducen el grado de paralelismo (máximo número de bloques en vuelo) ya que para una matriz de 10.000 columnas apenas encontramos 78 bloques en cada fila de la matriz si los bloques tienen

128 columnas.

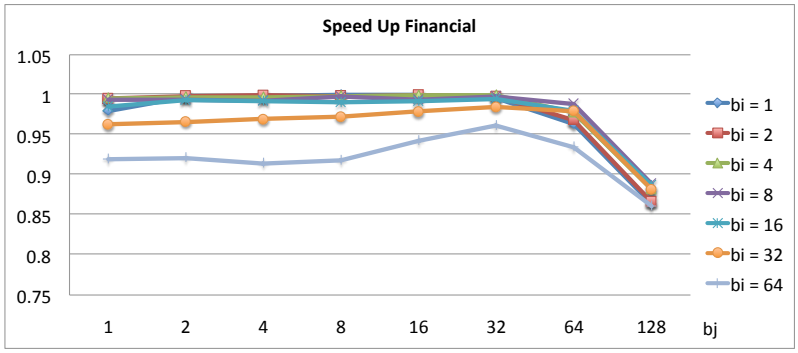



Figura 4.31: Speedup normalizado para el problema Financial en 32 cores y todos los tamaños de bloque, $bi \times bj$, válidos.

Comparando con el problema Smith-Waterman, claramente Financial presenta mucha mejor escalabilidad. Esto se debe a que para cada celda de la matriz hay que computar una suma que de media recorre $(n-1)/2$ elementos de la fila anterior. Es decir, este problema exhibe mucha más carga computacional que Smith-Waterman, y el impacto del tiling es por este motivo menos significativo.

Recordemos que la versión original sin optimizar ni siquiera alcanzaba un speedup de 3,5 (ver figura 4.1). El motivo de la mejora en rendimiento en este problema no es debido sólo al tiling, sino fundamentalmente a la optimización de simplificación de dependencias explicada en la sección 4.4.

#### 4.6.1.3. Checkerboard

En la figura 4.32 mostramos el speedup que medimos para el problema Checkerboard con distintos tamaños de $bi \times bj$ y ejecutando de 1 a 32 cores. El tamaño de la matriz es de $40.000 \times 40.000$ y todos los bloques son del tipo $1 \times bj$ ya que, como se explicó con anterioridad, para $bi > 1$ se provocan ciclos de dependencias. En la figura se aprecia que para 32 cores encontramos un punto de inflexión para $1 \times 256$ en el que el speedup deja de crecer. A partir de este punto el número de bloques independientes no es suficiente para mantener los 32 cores ocupados. Sin embargo, para 16 cores o menos, tamaños de bloque de incluso $1 \times 1024$ siguen siendo recomendados.

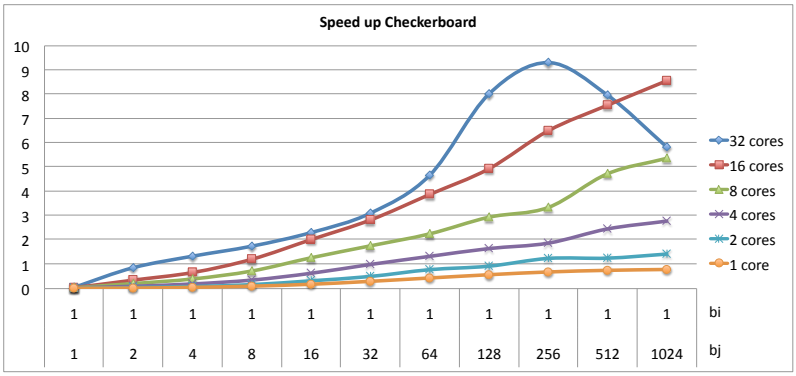



Figura 4.32: Speedup del problema Checkerboard para distintos tamaños de bloque $1 \times bj$ y distinto número de cores.

Nuevamente nos encontramos ante un problema de poca carga computacional. Para cada celda de la matriz sólo hay que calcular el mínimo de tres elementos de la fila superior y una suma con la función de coste. Por tanto, este es también un problema con poca intensidad aritmética y al igual que Smith-Waterman no sorprende que el problema no escale bien. Por ejemplo, para 4 cores la eficiencia es ya inferior a 75 %, y aún peor para 32 cores donde el speedup es sólo de 9.3, lo que resulta en una eficiencia del 30 %.

De cara a justificar el algoritmo de búsqueda del tamaño de bloque recomendado, y revisando de nuevo las figuras 4.29, 4.31, y 4.32, se observa que fijado $bi$, las curvas de speedup sólo presentan un punto de inflexión cuando varía el valor de $bj$. Aunque no representado en estas figuras, se observa el mismo comportamiento si se fija $bj$ y varía $bi$. De esta forma, tiene sentido que nuestro heurístico de búsqueda del tamaño de bloque recomendado vaya tomando muestras de los tiempos de ejecución buscando un punto de inflexión, como se explicó en la sección 4.3.4.

### 4.6.2. Coste de la búsqueda automática

En el apartado anterior ha quedado patente el impacto que puede tener implementar tiling, especialmente en problemas wavefront de grano fino. En esta sección evaluamos si la mejora en rendimiento compensa el tiempo de búsqueda del tamaño de bloque recomendado. Las preguntas que respondemos son las siguientes:

1. ¿Cuál es el tamaño de bloque encontrado y qué tiempo consume el código con dicho tamaño de bloque?

2. ¿Cuál es el tamaño de bloque óptimo que encuentra una búsqueda exhaustiva y qué tiempo consume ahora el código wavefront ejecutando con este tamaño de bloque óptimo?

3. ¿Hemos amortizado el tiempo invertido en la búsqueda del tamaño de bloque recomendado?. Es decir, cuanto tiempo nos ahorramos cuando el proceso consiste en primero buscar el tamaño de bloque y luego ejecutar con ese tamaño de bloque frente a ejecutar directamente el código sin tiling.

En la tabla 4.1 se muestran los parámetros que permiten responder a estas preguntas.

| | Smith-Waterman | Financial | Checkerboard |
|---|---|---|---|
| $bi \times bj$ **recomendado** | $32 \times 8$ | $8 \times 4$ | $1 \times 256$ |
| Tiempo de búsqueda | 0,8 seg. | 21 seg. | 0,3 seg. |
| T. ejecución con $bi \times bj$ recom. | 1,53 seg. | 71,1 seg. | 1,68 seg. |
| Speedup con $bi \times bj$ recom. | 12,66 | 27,5 | 9,3 |
| $bi \times bj$ **óptimo** | $16 \times 64$ | $1 \times 16$ | idem |
| Tiempo de búsqueda | 11,75 seg. | 1h15' | 1,38 seg. |
| T. ejecución con $bi \times bj$ óptimo. | 1,44 seg. | 70,9 | idem |
| Speedup con $bi \times bj$ óptimo | 13,49 | 27,4 | idem |
| Tiempo sin tiling | 155,41 seg. | 595,03 seg. | 330,82 seg. |

Tabla 4.1: Aspectos a considerar con respecto a los costes del tamaño de bloque recomendado.

Como se puede observar en la tabla, para el problema Checkerboard, la búsqueda ortogonal y la exhaustiva terminan encontrando el mismo tamaño de bloque. Sin embargo para los problemas Smith-Waterman y Financial no ocurre igual. Para Smith-Waterman, el tamaño de bloque recomendado es $32 \times 8$, mientras que el tamaño de bloque óptimo es $16 \times 64$. Para Financial el tamaño recomendado es el $8 \times 4$ y el que obtiene mejor rendimiento es el $1 \times 16$ . Sin embargo, la diferencia entre los tiempos de ejecución y speedup para ambos problemas para el tamaño de bloque recomendado y óptimo son asumibles, sobre todo teniendo en cuenta que el tiempo de búsqueda se divide por 14 en Smith-Waterman mientras que para Financial la búsqueda exhaustiva consume una hora y cuarto mientras que la búsqueda ortogonal devuelve el mismo resultado en sólo 21 segundos (214 veces más rápido).

Para el problema Checkerboard no sólo se encuentra el mismo tamaño de bloque independientemente del algoritmo de búsqueda utilizado sino que además el ahorro en tiempo de búsqueda puede ser bastante significativo. Para Checkerboard la búsqueda ortogonal es 4,6 veces más rápida que la búsqueda exhaustiva ya que en este caso el espacio de búsqueda es menor al requerirse que $bi = 1$ para que no haya ciclos de dependencias.

En cualquier caso, el tiempo invertido en la búsqueda de un tamaño de bloque recomendado está más que amortizada, ya que como vemos en la última fila de la tabla 4.1 el tiempo de ejecución de los códigos sin usar la plantilla optimizada y sin tiling es mucho mayor que el tiempo invertido en la búsqueda + el tiempo de ejecución optimizado.

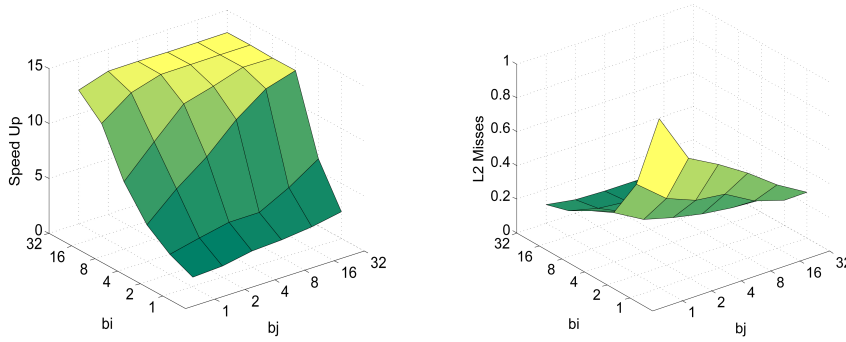
De cualquier modo, es fácil modificar la plantilla para que i) use el algoritmo exhaustivo para encontrar el tamaño óptimo de bloque, o ii) tomar las muestras del tiempo de ejecución para una fracción mayor del tiempo de ejecución o incluso para el tiempo de ejecución total del problema. Teniendo en cuenta que la búsqueda del tamaño de bloque óptimo no es una operación que haya que hacer frecuentemente, un usuario podría preferir invertir más tiempo en la búsqueda para tener una mayor seguridad de que en sucesivas ejecuciones está usando el mejor tamaño de bloque posible. Por tanto, estas dos opciones comentadas, que potencialmente pueden aumentar la precisión de la búsqueda también se consideran en la plantilla.

### 4.6.3. Fallos de caché e intensidad aritmética

En esta sección estudiamos el impacto de la latencia de acceso a memoria en el rendimiento de los códigos wavefront. Nos centramos en los problemas Smith-Waterman y Financial, el primero representando a un tipo de problemas con pocos accesos a memoria y pocas operaciones aritméticas por cada celda de la matriz, mientras que el segundo tiene mayor carga computacional y también hace más referencias a memoria por cada elemento de la matriz. El problema Checkerboard tiene un comportamiento similar a Smith-Waterman en este estudio. Dado que los problemas de estos dos códigos se agravan cuando aumenta el número de cores, el siguiente estudio asume las medidas en 32 cores. En particular, representaremos en figuras 3D el speedup y el número de fallos en la caché L2. Para las dos últimas magnitudes nos apoyamos en la librería PAPI que da acceso a los contadores HW del procesador Xeon® evaluado en este trabajo. Recordemos que nuestra arquitectura se basa en cuatro procesadores Intel Xeon® X7550 con 8 cores cada uno. Cada core dispones de cachés separadas de instrucciones y datos de nivel 1, L1D y L1I, de 32KB cada una, con líneas de 64 bytes y asociatividad 8. La caché de nivel dos unificada, L2, es también privada para cada core y de 256 KB (mismo tamaño de línea y asociatividad). Sin embargo la caché de nivel 3, está compartida por todos los cores y es de 18MB (16MB para la zona de almacenamiento y 2MB para el directorio caché).

La intensidad aritmética se define como el número de operaciones (enteras o punto flotante) que ejecuta un código por cada palabra transferida desde memoria. En otras palabras, la intensidad aritmética caracteriza cuando un problema está limitado por computación o por memoria. Si en un algoritmo la intensidad aritmética es baja, no se amor-

tiza el alto coste que implica la transferencia de los operandos desde memoria. Cuando la latencia con memoria es el cuello de botella del sistema, tener una alta intensidad aritmética es crucial. En las arquitecturas modernas esto es así para los problemas que no exhiben alta localidad (ni espacial ni temporal), ya que las caches internas no pueden ocultar el coste de todos los accesos a memoria lo que limita la escalabilidad del problema.




(a) (b)

Figura 4.33: Para el problema Smith-Waterman en 32 cores y distintos tamaños de bloque $bi \times bj$: (a) speedup, (b) fallos de cache L2.

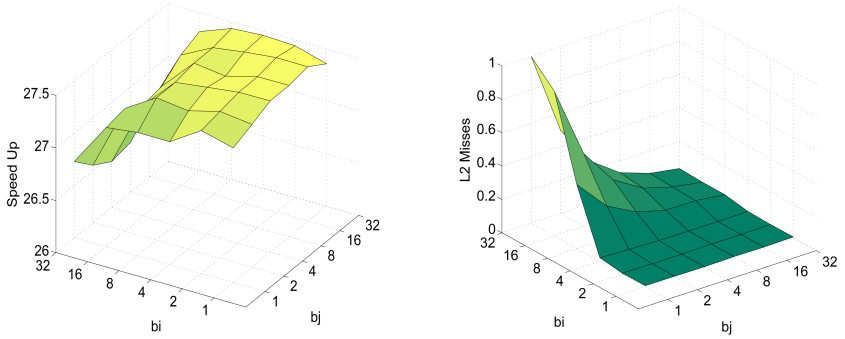
En las figuras 4.33(a) y 4.33(b) mostramos el speedup y el número normalizado de fallos en la cache L2 para distintos tamaños de bloque $bi \times bj$ y 32 cores. Dado que el número de instrucciones aritméticas es constante e independiente del tamaño de bloque, la intensidad aritmética se puede calcular como la ratio entre el número total de instrucciones aritméticas ejecutadas en el core y el número de fallos en la caché L2 que nos dá una estimación proporcional al número de accesos a memoria principal. Realmente, los accesos a memoria principal se pueden contabilizar mediante el número de fallos en la "Last Level Cache", LLC, pero dado que nuestra cache L3 es compartida y el contador HW reporta los fallos de caché de todos los cores, los resultados se desvirtúan. De esta forma, la gráfica correspondiente a la intensidad aritmética es especular (inversamente proporcional) al de fallos de caché L2.

Como se observa en la figura, a medida que aumenta el tamaño de bloque, $bi \times bj$, disminuye el número de fallos y por tanto aumenta la intensidad aritmética y también el speedup. El problema aparece cuando a partir de cierto tamaño de bloque, el número de fallos se estabiliza, la intensidad aritmética se aplana y por tanto el speedup deja de crecer.

Otro resultado interesante es que para valores bajos de $bi$, incrementar esta dimen-

sión del tamaño de bloque tiene mayor impacto en la reducción del número de fallos y por lo tanto en el aumento de la speedup. Esto ocurre gracias a la tecnología "Data Prefetch Logic", DPL, implementada para la cache L2 de este procesador perteneciente a la microarquitectura *Nehalem*. DPL implementa prefetch por hardware cuando se detecta un *stream* de datos lo que resulta en que para Smith-Waterman los accesos a cada línea, provoquen la precarga de la línea siguiente. Dado que el reciclado de tareas en nuestra plantilla prioriza que un mismo thread se encargue de procesar el bloque siguiente de la matriz, gracias a la precarga se garantizará que hayamos ocultado parte de la latencia de acceso a memoria principal. Esto explica porque incrementar $bj$ tiene menos impacto que incrementar $bi$ en la reducción del número de fallos. Sin embargo, dada la dependencia `(1,0)` cada celda necesita leer el valor de la celda inmediatamente superior (en la fila anterior). Con $bi = 2$, la segunda fila de cada bloque acierta en L2 cuando accede al primera. Para $bi = 4$, son las filas segunda, tercera y cuarta las que se benefician de aciertos en L2. Igualmente, para $bi = 8$ hay 7 filas que aciertan y así sucesivamente. Desde otra perspectiva, para $bi = 8$, de ocho filas, ha generado fallos de cache sólo una (la primera). Seguir aumentando $bi$ proporciona cada vez menores beneficios ya que el impacto de los fallos de una fila respecto de 16 o 32 filas no es tan representativo. Para un número más pequeño de cores el rendimiento se comporta de manera similar.

En las figuras 4.34(a) y 4.34(b) representamos también el speedup y el número de fallos normalizado pero ahora para el problema Financial, 32 cores y distintos tamaños de bloque.




(a) (b)

Figura 4.34: Para el problema Financial en 32 cores y distintos tamaños de bloque: (a) speedup, (b) fallos de cache L2.

En primer lugar, vemos que el speedup, figura 4.34(a), no depende con tanta intensidad del tamaño de bloque, ya que para todos los mostrados en la figura el speedup sólo

varía entre 26,75 y 27,5. El hecho de que el tamaño de bloque no afecte demasiado al speedup del problema Financial ya se ha comentado anteriormente y recordamos que se debe a que este problema si presenta un tamaño de grano grueso para cada tarea. Recordemos que cada celda de una fila de la matriz tiene que acceder a una media de $m/2$ elementos de la fila superior (5.000 en este experimento). Aún así, el tamaño de bloque tiene cierto impacto ya que puede limitar el grado de paralelismo del problema si no hay suficientes bloques, o como vemos en la figura 4.34(b) puede dar lugar a mas fallos de caché. En este caso vemos como los tamaños de bloque que dan lugar a más speedup son también los que generan menor número de fallos en la caché L2. Como ocurriera en Smith-Waterman, el comportamiento para un número inferior de cores es similar al comportamiento en 32 cores.

En este problema, vemos como el aumento progresivo del tamaño de $bj$ no tiene efecto en el número de fallos, de nuevo gracias a la prebúsqueda hardware. Sin embargo, que $bi$ crezca por encima de 4 si tiene un impacto en la efectividad de la caché L2. Creemos que esto se explica por un aumento de los fallos de capacidad ya que los elementos finales de cada fila tienen que acceder a casi toda la fila superior y cuando un bloque tiene más filas necesita que residan todas las filas superiores en caché para que no haya fallos.

La intensidad aritmética en este problema también es mayor que la que presenta el problema Smith-Waterman. Esto es así porque para cada celda de la matriz se calcula una función de interés una media de $m/2$ veces y en cada llamada se realizan varias operaciones aritméticas. Este tiempo consumido en las operaciones aritméticas oculta o solapa el tiempo necesario para las lecturas de memoria principal. Además la caché L3 compartida de 16MB y con política inclusiva (los datos en L1 o L2 están respaldados por una copia en L3) es capaz de contener la zona de la matriz a la que los 8 threads que se ejecutan dentro del procesador están accediendo durante cada momento de la ejecución. Esto se ha validado comprobando que el número de fallos en la caché L3 es pequeño en comparación con el número de accesos a la matriz. Por lo tanto este problema está menos limitado por memoria que el anterior, y como hemos comentado, esta es una de las razones para conseguir mayor escalabilidad.

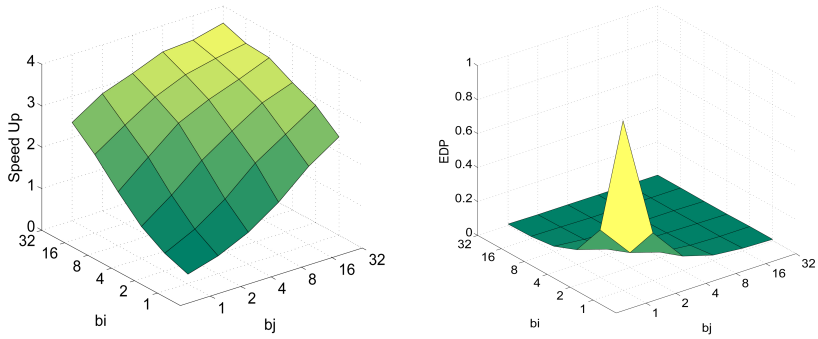
### 4.6.4. Relación entre eficiencia y consumo de energía

Las consideraciones respecto al consumo energético durante la ejecución de un código en paralelo son, desde hace unos años, un aspecto a tener en cuenta. En esta sección, queremos completar los resultados experimentales con un estudio que nos permita encontrar en qué situaciones nuestro problema paralelo es mas eficiente energéticamente hablando. Dado que no disponemos de las herramientas necesarias para medir el consu-

mo de energía en un core de nuestra arquitectura de 32 cores, no hemos podido llevar a cabo este estudio en dicha plataforma. Sin embargo, tenemos acceso a una arquitectura de la familia Ivy Bridge de Intel en la que es posible medir el consumo de energía gracias a contadores HW al efecto. Mas precisamente el procesador es un i5-3550 con 4 cores a 3.3GHz, corriendo bajo un sistema operativo Suse Linux. De nuevo hemos seleccionado los códigos Smith-Waterman y Financial compilados con el compilador icc y la opción de optimización -O3. Dado que la máquina tiene 4 cores el número máximo threads utilizados es también 4.

Las arquitecturas Ivy Bridge de Intel tienen contadores que se incrementan con distintos eventos del sistema [41]. Entre ellos disponemos de contadores que nos informan acerca del consumo de potencia. Para poder acceder a estos contadores utilizamos la librería *Performance Counter Monitor* (PCM) de Intel [42] que proporciona rutinas C++ y utilidades para estimar la utilización de los recursos del sistema. A diferencia de otros frameworks para leer contadores como PAPI, PCM permite recolectar datos *uncore*. Uncore es la parte del procesador que contiene el controlador de memoria integrada y la conexión a los otros procesadores y al hub E/S. Una de las utilidades que posee esta herramienta es el PCM-Power utilizada para medir el consumo de energía. En concreto, podemos medir el consumo de energía (en julios) empleado por los cores, la memoria y el total del sistema. Sin embargo, en nuestros experimentos optamos por recolectar el consumo energético en todo el sistema. El interfaz de utilización es muy parecido a la herramienta PAPI de manera que tan solo debemos leer los contadores antes y después del fragmento de código que queremos medir. Por tanto, los contadores del i5 que hemos medido nos devuelven la energía total consumida en los 4 cores. Pero además de la energía hemos querido también considerar otra magnitud, el producto de la energía por el tiempo de ejecución. Esta medida, en inglés llamada EDP, "Energy-Delay Product" involucra en una única magnitud dos aspectos que queremos minimizar: la energía consumida y el tiempo de ejecución.

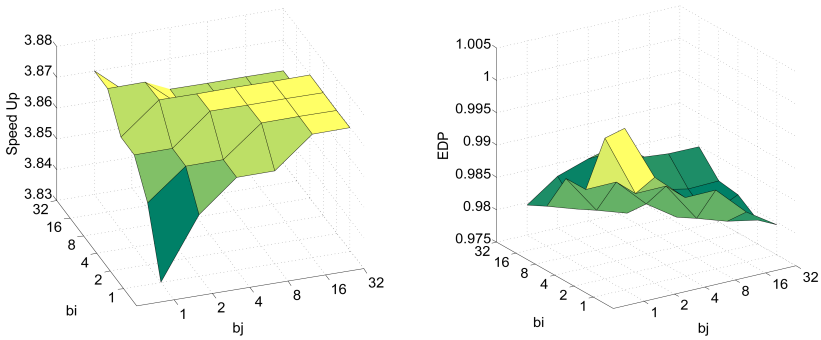
En este estudio, hemos medido los problemas Smith-Waterman y Financial en 4 cores y con distintos tamaños de bloque para ver que tamaño de bloque produce un menor consumo y la relación del consumo con el rendimiento. Empezando con el problema Smith-Waterman, en la figura 4.35 vemos el speedup y el EDP normalizado para distintos tamaños $bi \times bj$. La gráfica del consumo energético en Julios no se muestra ya que tiene exactamente la misma forma que la que mostramos con el EDP. Vemos como en esta arquitectura este problema consigue las mejoras eficiencias cuando el tamaño de bloque aumenta hasta $32 \times 32$.




(a) (b)

Figura 4.35: Para el problema Smith-Waterman en 4 cores y distintos tamaños de bloque: (a) speedup, y (b) EDP normalizado.

De nuevo vemos como el tamaño de bloque tiene un gran impacto en el speedup para este problema, ya que hay grandes diferencias de esta magnitud para los distintos $bi \times bj$. También se aprecia que para el tamaño de bloque $1 \times 1$ es para el que se consigue no sólo peor speedup sino también mayor energía y EDP. La mínima energía, 112 Julios, se consume para el tamaño de bloque $32 \times 32$. Por tanto se cumple en este caso que un tamaño de bloque que propicia que el problema se resuelva antes también da lugar a que el consumo energético sea menor.




(a) (b)

Figura 4.36: Para el problema Financial en 4 cores y distintos tamaños de bloque: (a) speedup, y (b) EDP normalizado.

Por último, en la figura 4.36 mostramos las gráficas para el problema Financial de

speedup y EDP normalizado, para 4 cores y distintos tamaños de bloque. Al igual que ocurría en la arquitectura Xeon de 32 cores, en esta nueva plataforma basada en i5, el tamaño de bloque no tiene demasiado impacto en el speedup: para todos los $bi \times bj$ mostrados, el speedup sólo cambia entre 3,83 y 3,87. Los mejores tamaños de bloque se encuentran para tamaños grandes de $bj$ y valores menores de 8 para $bi$. La superficie del consumo de Julios es prácticamente igual a la que mostramos para el EDP en la figura 4.36. El valor mínimo de energía es de 9210 Julios para el tamaño $2 \times 32$. Vemos que el rango de variación del EDP también es muy pequeño, estando el valor del EDP máximo normalizado en 1 y el mínimo en 0,98. Aún así, de nuevo se aprecia la correlación entre speedup y consumo energético, ya que los tamaños de bloque para los que el problema consigue mejor speedup son los mismos para los que obtenemos un menor consumo energético y EDP.

## 4.7. Trabajos relacionados

Como comentamos en el capítulo anterior, existen muchos trabajos que estudian los problemas de tipo wavefront. Pero de ellos no hay muchos en los que se haga especial énfasis en el tiling y el cálculo del tamaño de bloque óptimo. En el trabajo [50], el autor propone un nuevo patrón de diseño llamado *block-cyclic based wavefront* que aplicado en sistemas de memoria distribuida obtiene un gran rendimiento y una alta eficiencia. Para explicar este patrón, el autor utiliza el problema Smith-Waterman realizando tiling sobre los elementos de la matriz para posteriormente distribuir los bloques en distintos procesadores según a que columna pertenezca cada bloque. De esta manera, utiliza tiling para obtener un grano de tarea más grande tal y como realizamos en nuestra plantilla. Nos diferenciamos de este trabajo en que nosotros proponemos una plantilla de alto nivel para realizar tiling automático de problemas wavefront abstrayendo al programador de la gestión manual del tiling y de la planificación de las tareas en los distintos procesadores, mientras que en [50] se propone un método para implementar problemas de tipo wavefront, donde el programador tiene que realizar el tiling manual y distribuir la carga de trabajo de forma estática en los distintos procesadores. En cuanto a los experimentos que expone, éstos están enfocados a arquitecturas distribuidas mientras que nosotros nos centramos en sistemas de memoria compartida.

Existen diversas aproximaciones para implementar problemas wavefront en sistemas heterogéneos. Por ejemplo en [25] se ofrece un compilador que haciendo tiling mejora dos niveles distintos de paralelismo en problemas de tipo wavefront paralelizando bucles anidados con dependencias uniformes en CUDA. El framework presenta dos optimizaciones claves: la primera encontrar un tiling en un hiperplano con balanceo intra-tile y eliminar algunas dependencias falsas mejorando el paralelismo entre bloques.

Por otro lado, en el trabajo [32] explican la importancia de la intensidad aritmética en los problemas stencil y proponen hacer tiling para aumentar la intensidad aritmética y la localidad temporal de la cache. Con el mismo objetivo en [48], la autora propone un esquema para realizar tiling sobre problemas stencil. Presentan un algoritmo teórico con coste polinomial que realizando un análisis de coste de memoria, calcula el tamaño óptimo de bloque. En nuestro caso, utilizamos pruebas empíricas para estimar el tamaño de bloque óptimo por lo que nuestros resultados se ajustan a la realidad del problema en cada arquitectura sin requerir un conocimiento a priori de la jerarquía de memoria. Nuevamente, las técnicas que se proponen en este trabajo tienen que ser implementadas manualmente por un programador, mientras que nuestra plantilla evita al usuario tener que trabajar con aspectos de diseño de bajo nivel.

En el trabajo [51] los autores proponen hacer tiling para problemas wavefront después de aplicar técnicas de skewing. En su artículo sostienen que un tamaño de bloque grande sólo tiene paralelismo suficiente para un número limitado de cores, y afirman que un tamaño de bloque pequeño produce un paralelismo suficiente tal y como hemos visto en este capítulo. Un tamaño de bloque pequeño, fomenta la localidad entre bloques perjudicando la localidad dentro de un mismo bloque. Ellos rechazan la planificación dinámica porque se asigna arbitrariamente los bloques a distintos procesadores. Sin embargo, utilizando nuestra plantilla podemos guiar al planificador para reciclarnos en el bloque que vaya a aprovechar de manera más eficaz la memoria cache. En su trabajo comparan una estrategia de planificación dinámica con una estática y de sus resultados concluyen que la planificación estática produce mejores speedup. Nosotros en nuestro trabajo realizamos una búsqueda ortogonal para encontrar el tamaño de bloque adecuado para cada problema en la arquitectura en la que se esté ejecutando. Generalmente, coincidimos con este trabajo en que se deben elegir bloques no muy grandes para evitar llegar a una situación de paralelismo insuficiente.

## 4.8. Conclusiones

En el capítulo anterior presentamos una plantilla para implementar problemas de tipo wavefront de manera productiva y eficiente. Utilizamos cinco benchmarks para medir su rendimiento, demostrando que utilizando la plantilla, los programadores pueden ahorrarse hasta un 50 % de esfuerzo, introduciendo tan solo un 5 % de overhead respecto a la implementación de los mismos benchmarks sin utilizar la plantilla. Sin embargo, encontramos problemas de escalabilidad en los problemas cuando el grano de tarea es fino o el overhead debido a la gestión de dependencias es prohibitivo. Por eso, en este capítulo hemos introducido unas optimizaciones en la plantilla que mejoran su eficiencia y productividad:

1. Encontramos que la inicialización de los contadores era un problema que solventamos inicializando tan solo los contadores estrictamente necesarios para el comienzo de la ejecución, para luego ir inicializando el resto de los contadores dinámicamente según vaya siendo necesario.

2. Realizamos tiling automático para aumentar el grano de tarea de los problemas. De este modo, logramos los siguientes objetivos: i) aprovechar mejor la localidad espacial del problema, y ii) disminuir el número de tareas, lo que conlleva una reducción del overhead introducido por la función `spawn` y por la gestión de los contadores. Introducir tiling puede provocar que existan ciclos que antes no existían por lo que hay que descartar los tamaños de bloque que provoquen ciclo, para lo cual hemos diseñado un algoritmo basado en el I-test.

3. Cuando el número de dependencias en un problema wavefront es muy grande, la actualización de los contadores puede requerir un tiempo equivalente o incluso superior al necesario para la propia computación del problema. Para solucionar este problema implementamos un algoritmo que simplifica el número de dependencias, dejando en el usuario la responsabilidad de aprovechar o no esta facilidad. Para el problema Financial, esta optimización es clave a la hora de aumentar la eficiencia del ejecutable.

4. Introducimos los mecanismos necesarios en la plantilla para que ésta realice una búsqueda ortogonal automática que termina devolviendo un tamaño de bloque recomendado. Puede que éste tamaño no sea el óptimo global, pero en nuestros experimentos ha resultado ser suficientemente bueno, sin perjuicio de que el usuario desee llevar a cabo una búsqueda más exhaustiva del tamaño de bloque óptimo.

Tras esto, comparamos las nuevas implementaciones de tres problemas reales, y los resultados confirman que la plantilla optimizada consigue un speedup entre 7x y 21x para los tres problemas, respecto a la versión basada en la plantilla no optimizada.

En resumen, en este capítulo hemos mejorado nuestra plantilla de manera que para el usuario siga siendo sencillo programar problemas de tipo wavefront (escribir un fichero de definición y especificar el trabajo de cada tarea); pero además ahora se consigue una mayor eficiencia.

# 5 Conclusiones

En este trabajo hemos abordado la paralelización basada en tareas de dos tipos de problemas, pipeline y wavefront, con el propósito de conseguir una mayor eficiencia en la ejecución al tiempo que facilitamos una implementación paralela basada en plantillas que simplifiquen el trabajo del programador. La programación paralela se ha convertido en un paradigma imprescindible para desarrollar aplicaciones que aprovechen bien los recursos que ofrecen los sistemas multicores. La paralelización basada en tareas ha ganado bastante popularidad recientemente y precisamente, los dos tipos de problemas citados se muestran especialmente aptos aprovechar este paradigma. Estos problemas de tipo pipeline y wavefront se pueden encuadrar en el marco de los problemas de tipo *stream*, en los que a cada elemento de una secuencia de datos (espaciados en el tiempo) se le aplica una serie de operaciones o funciones. A continuación resumimos las conclusiones de cada uno de los apartados de esta memoria, terminando con una exposición sobre las posibles líneas futuras de trabajo.

## 5.1. Elección del modelo de programación

Como primer paso para la consecución del objetivo descrito en la primera frase de este capítulo, hemos querido verificar que el modelo de programación basado en tareas es adecuado para mejorar la productividad de los problemas estudiados. Tras los experimentos previos realizados se concluyó que en comparación con un modelo de threads, **el modelo basado en tareas es ventajoso**, principalmente por las siguientes razones:

- Las tareas son más ligeras que los threads y por tanto la creación/terminación/-planificación de threads es más lenta que la creación/terminación/planificación de

tareas. Adicionalmente, trabajando con threads podemos incurrir en situaciones de *oversubscription* o *undersubscription* con los consiguientes overheads asociados.

- El planificador de tareas puede ser guiado con información de alto nivel. Además, contrariamente al planificador de threads del sistema operativo, el planificador de tareas no tiene otro objetivo que no sea maximizar la eficiencia. Este planificador es capaz, en la mayoría de los casos, de balancear automáticamente la carga computacional mediante una estrategia de work-stealing.

- En cuanto a la productividad, en el modelo basado en tareas el programador puede centrar la atención en el flujo de datos y/o flujo de control de los algoritmos, sin preocuparse por detalles de más bajo nivel como el de la gestión directa de los threads. De esta forma se gana en simplicidad tanto de la implementación como del mantenimiento de los códigos.

El éxito de los modelos basados en tareas es más evidente cuando se enumeran los lenguajes paralelos o librerías que dan soporte a este paradigma. Sin ser exhaustivos, entre ellos encontramos OpenMP 3.0 o superior, CnC, Cilk, Intel Threading Building Blocks (TBB), java.util.concurrent de Java, Microsoft TPL (*Task Parallel Library*) incluida en .Net 4.0, Nanox RT, Qthreads, Chapel o X10.

## 5.2. Modelo basado en tareas para problemas de tipo pipeline.

Hemos validado el modelo basado en tareas para el patrón pipeline, utilizando dos benchmarks del conjunto PARSEC (ferret, una aplicación para la búsqueda de similitudes de imágenes; y dedup, una aplicación para comprimir un stream de datos). Las versiones originales de estos códigos están implementadas con Pthreads y presentan dos problemas que limitan su rendimiento: el desbalanceo de carga y el cuello de botella de entrada/salida. Para corregir el problema de desbalanceo, implementamos ferret y dedup aplicando las soluciones propuestas en [57]: i) colapsar varias etapas en una sola; y b) recurrir a la planificación dinámica basada en work-stealing mediante TBB. La implementación en TBB es directa para ferret, pero no para dedup, ya que este problema dispone de una etapa que recibe un elemento de entrada y devuelve un número indeterminado de elementos de salida. Dado que en la actualidad TBB no ofrece soporte a este tipo de filtro complejo, como primera aproximación se opta por una implementación híbrida, que combina TBB con Pthreads. Comparando las nuevas versiones con las originales se demuestra que las implementaciones en TBB superan a las basadas en Pthreads.

A continuación y centrándonos en dedup, proponemos un nuevo filtro TBB que permita implementar este problema sin necesidad de usar threads. Más precisamente modificamos la plantilla pipeline de TBB para incluir un nuevo tipo de filtro, llamado **Multioutput**, que permita una relación uno a muchos entre el número de elementos de entrada al filtro y el número de elementos de salida del mismo. Además de la implementación de dedup basada en el nuevo filtro y de la versión híbrida, también evaluamos otra posibilidad basada en pipeline anidados. La evaluación experimental de las tres implementaciones concluye que con el filtro Multioutput no sólo se reduce el overhead sino que también se facilita la codificación del problema. Además, se complementa el estudio elaborando un modelo basado en teoría de colas que permite comparar de forma analítica los tres diseños.

## 5.3. Modelo basado en tareas para problemas de tipo wavefront

De entre los códigos listados como de tipo pipeline en la suite PARSEC, además de ferret y dedup, también se encuentra un código H264. Sin embargo, en realidad este código presenta un patrón de tipo wavefront, así que abordamos también este tipo de problemas desde la perspectiva del modelo de programación basado en tareas.

En primer lugar, para este patrón wavefront se estudia cuál de los modelos basados en tareas proporciona más rendimiento. Evaluamos un problema wavefront implementado con TBB, Cilk, CnC y OpenMP 3.0 y considerando distintas alternativas de diseño. El resultado de la comparación es que TBB ofrece el mejor rendimiento, gracias, sobre todo, a la posibilidad de reciclar tareas reduciendo overheads y a priorizar un recorrido de los datos que explote mejor la jerarquía de memoria.

Así como para problemas de tipo pipeline existe una plantilla TBB al efecto, para problemas de tipo wavefront no existe una solución equivalente y es necesario implementar estos códigos recurriendo directamente a los métodos de bajo nivel que soportan tareas en TBB. Para subsanar esta carencia proponemos una plantilla que **simplifica la paralelización de problemas de tipo wavefront**. Esta plantilla encapsula toda la gestión de tareas y de dependencias entre ellas. El usuario tan sólo tiene que proporcionar un fichero con el patrón de dependencias e implementar el código que ha de ejecutarse para cada elemento de la matriz.

En relación con dicha plantilla:

- Para validar nuestra plantilla, hemos estudiado cuatro benchmarks de problemas reales, entre ellos el H264. Los resultados arrojan que la plantilla sólo incurre en

un overhead adicional inferior al 5 % respecto de la implementación a bajo nivel en TBB basada en gestión manual de tareas. Además, la versión H264 implementada con nuestra plantilla es mejor que la versión original de PARSEC basada en Pthreads, debido a que no hay que gestionar las colas de tareas y a un mejor balanceo de la carga.

- También cuantificamos en qué medida se simplifica la labor de programación gracias a la plantilla, comprobando que recurrir a la misma supone una reducción del esfuerzo del programador de entre un 25 % y un 50 %, comparando con una implementación manual.

Por lo tanto, podemos concluir que **nuestra plantilla es una herramienta efectiva, con un bajo coste en términos de overhead, y altamente productiva para el programador**.

Sin embargo, para problemas de grano muy fino las implementaciones de algunos de los códigos no escalan adecuadamente. En estos casos las fuentes de overhead recaen en la inicialización y actualización de los contadores atómicos que mantienen la información de dependencias. Para corregir esta situación se proponen tres optimizaciones al funcionamiento interno de la plantilla de wavefront:

- En el arranque del problema inicializar sólo los contadores imprescindibles y retrasar el resto de las inicializaciones al momento en que sean necesarias. De esta forma la inicialización se solapa con la ejecución del problema y se implementa de forma paralela.

- Recurrir a una estrategia de tiling que reduce el número de contadores y de tareas. De esta forma no sólo se reduce el overhead de creación de tareas sino que también se intenta explotar mejor la localidad. La plantilla no sólo es capaz de generar automáticamente una versión con tiling del problema wavefront, sino que también es capaz de encontrar en poco tiempo un tamaño recomendado para el tile.

- Por último, también se propone un mecanismo de trasnformación de dependencias que reduce el número de las mismas. En la mayoría de los casos, esta simplificación no afecta al rendimiento de la ejecución paralela del problema, y si lo hace puede ser desactivada por el usuario.

También realizamos un estudio del consumo energético y del impacto de la intensidad aritmética en el rendimiento del problema. Se comprueba que los tamaños de tile que proporcionan mayor speedup son los que también consiguen menor consumo energético y menor número de fallos de caché.

## 5.4. Trabajos futuros

Durante el desarrollo de este trabajo se han encontrado posibles líneas de investigación que proponemos abordar en el futuro:

- Desde hace unos años es patente el gran interés que despiertan los trabajos basados en GPUs. Los problemas de tipo wavefront no han escapado de esta corriente y hay varias implementaciones para GPU de problemas como Smith-Waterman. Por poner un sólo ejemplo, en [61] se contemplan varias técnicas para paralelizar problemas de este tipo en GPUs utilizando CUDA. En esta línea, queremos incorporar a nuestra plantilla la posibilidad de implementar el patrón wavefront en sistemas heterogéneos. El objetivo sería conseguir una distribución automática de carga entre todos los recursos del sistema, considerando cores y GPUs. Ya hemos dado un primer paso en ese sentido, pero los resultados usando GPUs discretas de la familia Nvidia Fermi revelan que las GPUs son lo suficientemente rápidas como para ejecutar todo el trabajo sin ayuda de los cores. Sin embargo, existen arquitecturas, como Ivy Bridge o Haswell, que poseen GPUs mucho más ligeras que sí se verían beneficiadas de la colaboración de los cores para reducir significativamente el tiempo de ejecución.

- Otro uso para el que podríamos extender la plantilla TBB es para implementar más problemas de tipo stencil. Algo parecido ya se hizo en el lenguaje ZPL [13] (precursor de Chapel [14]), pero el compilador de ZPL está discontinuado y Chapel aún no tiene dicha funcionalidad. Dentro de los problemas stencil, un primer paso podría ser abordar variantes de tipo Gauss-Seidel con patrones de dependencias más complejas como la que resultan en esquemas tipo “time skewing” [60]. En este contexto el tiling puede tener un mayor impacto ya que estos códigos suelen exhibir una mayor localidad temporal. Por otro lado, estos problema sufren de mayor desbalanceo de carga entre los distintos bloques, aspecto que podría aliviarse usando un planificador basado en work-stealing como el de TBB.

- Otro trabajo interesante podría consistir en abordar los problemas de tipo wavefront pero esta vez en arquitecturas de memoria distribuidas. Para ello, recorreríamos un camino similar al que hemos andado para arquitecturas de memoria compartida: primero estudiar que lenguajes o librerías para este tipo de arquitecturas (MPI [30], GASNet [10], Chapel [14], UPC [18]) proporciona más ventajas para posteriormente adaptar nuestra plantilla al modelo que obtenga mejor rendimiento.

- Por último, en aras de mejorar la productividad, nos gustaría simplificar el interfaz que ofrecemos al usuario de la plantilla wavefront. Una posibilidad sería propor-

cionar un Graphical User Interface (GUI) que permita “dibujar” directamente los vectores de dependencias, evitando al usuario tener que editar un fichero de definición en modo texto.

# Bibliografía


[1] Ashwin M. Aji, Wu-chun Feng, Filip Blagojevic, and Dimitrios S. Nikolopoulos. Cell-swat: modeling and scheduling wavefront computations on the cell broadband engine. In *CF '08: Proceedings of the 5th conference on Computing frontiers*, pages 13–22, New York, NY, USA, 2008. ACM. (Cited on page 140)

[2] Farhana Aleen, Monirul Sharif, and Santosh Pande. Input-driven dynamic execution prediction of streaming applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 315–324, 2010. (Cited on page 76)

[3] Arnold O. Allen. *Probability, Statistics, and Queueing Theory - With Computer Science Applications*. Academic Press; 2 edition, 1990. (Cited on pages 54, 55, 56 and 60)

[4] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. Generating parallel programs from the wavefront design pattern. *Parallel and Distributed Processing Symposium, International*, 2:0104, 2002. (Cited on pages 8, 121 and 140)

[5] Utpal Banerjee. *Speedup of ordinary programs*. Ph.D. thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, Urbana-Champaign, October 1979. (Cited on page 160)

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008. (Cited on pages 25, 34 and 36)

[7] Lawrence Livermore National Laboratory Blaise Barney. *POSIX Threads Programming*. https://computing.llnl.gov/tutorials/pthreads/. (Cited on page 5)

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995. (Cited on page 30)

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. (Cited on page 23)

[10] Dan Bonachea. Proposal for extending the UPC memory copy library functions and supporting extensions to GASNet, Version 2.0. Technical report, LBNL, 2007. (Cited on page 211)

[11] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM. (Cited on pages 24 and 76)

[12] Miguel Á. Carreira-Perpiñán. Fast nonparametric clustering with gaussian blurring mean-shift. In *Proceedings of the 23rd international conference on Machine learning*, ICML '06, pages 153–160, New York, NY, USA, 2006. ACM. (Cited on page 6)

[13] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211, March 2000. (Cited on page 211)

[14] Cray Chapel. *The Chapel Programming Language*. http://chapel.cs.washington.edu. (Cited on pages 1 and 211)

[15] Matthias-Michael Christen. *Generating and Auto-Tuning Parallel Stencil Codes*. PhD thesis, Affoltern BE, Schweiz, 2011. (Cited on page 7)

[16] *A parallel implementation of the Smith-Waterman algorithm for massive sequences searching*, volume 2, 2004. (Cited on page 120)

[17] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 536–545, Washington, DC, USA, 2008. IEEE Computer Society. (Cited on page 23)

[18] UPC Consortium. UPC language spec, v1.2. Technical report, LBNL-59208, 2005. (Cited on pages 1 and 211)

[19] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57–66, Sept. 2008. (Cited on pages 13, 17 and 30)

[20] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, March 2004. (Cited on page 24)

[21] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society. (Cited on page 76)

[22] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for stream processing. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 33–42, New York, NY, USA, 2006. ACM. (Cited on page 76)

[23] Vazirani Umesh Dasgupta Sanjoy, Papadimitriou Christos. *Algorithms*. McGraw-Hill Higher Education, 2007. (Cited on page 122)

[24] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press. (Cited on page 5)

[25] Peng Di, Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. Automatic parallelization of tiled loop nests with enhanced fine-grained parallelism on gpus. In *ICPP*, pages 350–359, 2012. (Cited on page 204)

[26] Antonio J. Dios, Rafael Asenjo, Angeles Navarro, Francisco Corbera, and Emilio L. Zapata. Evaluation of the task programming model in the parallelization of wavefront problems. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communications*, HPCC '10, pages 257–264, 2010. (Cited on page 46)

[27] Antonio J. Dios, Rafael Asenjo, Angeles Navarro, Francisco Corbera, and Emilio L. Zapata. High-level template for the task-based parallel wavefront pattern. *High-Performance Computing, International Conference on*, 0:1–10, 2011. (Cited on page 145)

[28] J. Falcou, J. Srot, T. Chateau, and J. Laprest. Quaff: Efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7–8):604–615, 2006. (Cited on page 140)

[29] *Project Fortress Overview*. http://projectfortress.sun.com. (Cited on page 1)

[30] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009. (Cited on page 211)

[31] Basilio B. Fraguela, Jia Guo, Ganesh Bikshandi, María J. Garzarán, Gheorghe Almási, José Moreira, and David Padua. The hierarchically tiled arrays programming approach. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–12, New York, NY, USA, 2004. ACM. (Cited on page 1)

[32] P Ghysels, P Klosiewicz, and W. Vanroose. Improving the arithmetic intensity of multigrid with the help of polynomial smoothers. *Numerical Linear Algebra and Applications*, 19:253–267, 2012. (Cited on page 205)

[33] Carlos H. Gonzalez and Basilio B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. *10th IEEE International Conference on High Performance Computing and Communications*, pages 79–88, 2010. (Cited on page 138)

[34] F. Guirado, A. Ripoll, C. Roig, and E. Luque. Exploitation of parallelism for applications with an input data stream: Optimal resource-throughput tradeoffs. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, PDP '05, pages 170–178, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 75)

[35] Fernando Guirado, Ana Ripoll, Concepció Roig, Xiao Yuan, and Emilio Luque. Predicting the best mapping for efficient exploitation of task and data parallelism. In *Euro-Par*, pages 218–223, 2003. (Cited on page 75)

[36] Jayanth Gummaraju, Joel Coburn, Yoshio Turner, and Mendel Rosenblum. Streamware: programming general purpose multicore processors using streams. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 297–307, New York, NY, USA, 2008. ACM. (Cited on page 76)

[37] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977. (Cited on page 138)

[38] Shane G. Henderson and Samuel M. T. Ehrlichman. Sharpening comparisons via gaussian copulas and semidefinite programming. *ACM Trans. Model. Comput. Simul.*, 22(4):22:1–22:21, November 2012. (Cited on page 6)

[39] P.N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium language ref. manual. Technical report, CS Division (EECS), UC, Berkeley, CA, 2001. (Cited on page 1)

[40] Amir H. Hormati, Yoonseo Choi, Manjunath Kudlur, Rodric Rabbah, Trevor Mudge, and Scott Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 214–223, 2009. (Cited on page 76)

[41] Intel. *Intel Ivy Bridge Microarchitecture events*, Apr 2010. http://oprofile.sourceforge.net/docs/intel-ivybridge-events.php. (Cited on page 202)

[42] Intel. *Intel® Performance Counter Monitor - A better way to measure CPU utilization*, Apr 2010. http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpu-utilization. (Cited on page 202)

[43] Intel Corporation. *Threading Building Blocks*. http://www. threadingbuildingblocks.org/. (Cited on pages 9 and 24)

[44] X. Kong, D. Klappholz, and K. Psarris. The i test: An improved dependence test for automatic parallelization and vectorization. *IEEE Trans. Parallel Distrib. Syst.*, 2(3):342–349, July 1991. (Cited on pages 158, 159 and 162)

[45] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 114–124, 2008. (Cited on page 76)

[46] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, 2009. (Cited on page 24)

[47] E Christopher Lewis and Lawrence Snyder. Pipelining wavefront computations: Experiences and performance. In *In Fifth IEEE International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, 1999. (Cited on pages 139 and 140)

[48] Zhiyuan Li and Yonghong Song. Automatic tiling of iterative stencil loops. *ACM Trans. Program. Lang. Syst.*, 26(6):975–1028, November 2004. (Cited on page 205)

[49] Wei-Keng Liao, Alok Choudhary, Donald Weiner, and Pramod Varshney. Performance evaluation of a parallel pipeline computational model for space-time adaptive processing. *J. Supercomput.*, 31(2):137–160, 2004. (Cited on page 76)

[50] Weiguo Liu and B. Schmidt. Parallel design pattern for computational biology and scientific computing applications. In *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 456 – 459, dec. 2003. (Cited on page 204)

[51] Naraig Manjikian and Tarek S. Abdelrahman. Scheduling of wavefront parallelism on scalable shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing ICPP 96*. CRC Press, 1996. (Cited on page 205)

[52] Sebastian Mattheis, Tobias Schuele, Andreas Raabe, Thomas Henties, and Urs Gleim. Work stealing strategies for parallel stream processing in soft real-time systems. In *Proceedings of the 25th international conference on Architecture of Computing Systems*, ARCS'12, pages 172–183, 2012. (Cited on page 76)

[53] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004. (Cited on page 3)

[54] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308 – 320, dec. 1976. (Cited on page 138)

[55] Mauricio Alvarez Mesa, Alex Ramirez, Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink, and Mateo Valero. Scalability of macroblock-level parallelism for h.264 decoding. *Parallel and Distributed Systems, International Conference on*, 0:236–243, 2009. (Cited on pages 130, 131 and 137)

[56] Shirley Moore, Dan Terpstra, Vince Weaver, Heike Jagode, James Ralph, and Jack Dongarra. Poster: new features of the papi hardware counter library. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, SC '11 Companion, pages 3–4, 2011. (Cited on page 68)

[57] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 281–290, 2009. (Cited on pages 26, 27, 29, 30, 31, 35, 36, 50, 54, 55, 56, 59, 64, 65, 77 and 208)

[58] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. (Cited on page 1)

[59] *The OpenMP API specification for parallel programming*. http://www.openmp.org/. (Cited on page 31)

[60] Nissa Osheim, Michelle Mills Strout, Dave Rostron, and Sanjay V. Rajopadhye. Smashing: Folding space to tile through time. In José Nelson Amaral, editor, *LCPC*, volume 5335 of *Lecture Notes in Computer Science*, pages 80–93. Springer, 2008. (Cited on page 211)

[61] S. J. Pennycook, G. R. Mudalige, S. D. Hammond, and S. A. Jarvis. Parallelising wavefront applications on general-purpose gpu devices, 2010. (Cited on page 211)

[62] *PETSc: Portable, Extensible Toolkit for Scientific Computation*. http://www.mcs.anl.gov/petsc/. (Cited on page 1)

[63] *Pipeline Processing*. http://www.cise.ufl.edu. (Cited on page 4)

[64] Bruno Richard Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. wiley, 2000. 635 pp. ISBN 0-471-34613-6. (Cited on pages 8 and 127)

[65] Shah M. Faizur Rahman, Qing Yi, and Apan Qasem. Understanding stencil code performance on multicore architectures. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 30:1–30:10, New York, NY, USA, 2011. ACM. (Cited on page 7)

[66] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*, pages 114–123, New York, NY, USA, 2008. ACM. (Cited on page 75)

[67] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 31)

[68] G. Redinbo. Queueing systems, volume i: Theory–leonard kleinrock. *Communications, IEEE Transactions on*, 25(1):178–179, Jan 1977. (Cited on page 55)

[69] Eric C. Reed, Nicholas Chen, and Ralph E. Johnson. Expressing pipeline parallelism using tbb constructs: a case study on what works and what doesn't. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, VMIL'11*, SPLASH '11 Workshops, pages 133–138, New York, NY, USA, 2011. ACM. (Cited on page 5)

[70] Eric C. Reed, Nicholas Chen, and Ralph E. Johnson. Expressing pipeline parallelism using tbb constructs: a case study on what works and what doesn't. In *SPLASH '11 Workshops*, pages 133–138, 2011. (Cited on pages 39, 51 and 77)

[71] James Reinders. *Intel Threading Building Blocks: Multi-core parallelism for C++ programming*. O'Reilly, 2007. (Cited on pages 1, 10, 17 and 30)

[72] Paul Rendell. Collision-based computing. In Andrew Adamatzky, editor, *Collision-based computing*, chapter Turing universality of the game of life, pages 513–539. Springer-Verlag, London, UK, UK, 2002. (Cited on page 7)

[73] Sean Rul, Hans Vandierendonck, and Koen De Bosschere. A profile-based tool for finding pipeline parallelism in sequential programs. *Parallel Comput.*, 36(9):531–551, September 2010. (Cited on page 75)

[74] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 22–32, 2011. (Cited on pages 24 and 76)

[75] Jaeho Shin. Measure memory usage of processes. git://gist.github.com/526585.git. (Cited on page 46)

[76] O. Storaasli and D. Strenski. Exploring accelerating science applications with fpgas. In *Proc. of the Reconfigurable Systems Summer Institute*, July 2077. (Cited on page 140)

[77] Y.c. Tay and HweeHwa Pang. Load sharing in distributed multimedia-on-demand systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):410–428, 2000. (Cited on pages 59 and 60)

[78] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the Compiler Construction*, France, 2002. (Cited on pages 24 and 76)

[79] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on*

*Microarchitecture*, pages 356–369, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 75)

[80] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209, 2009. (Cited on page 76)

[81] Antonio Vallecillo and Rosa Guerequeta. *Técnicas y diseño de algoritmos*. Universidad de Malaga, 1998. (Cited on page 8)

[82] Zheng Wang and Michael F.P. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 307–318, 2010. (Cited on page 76)

[83] Niklaus Wirth. Tasks versus threads: An alternative multiprocessing paradigm. *Software - Concepts and Tools*, 17(1):6–12, 1996. (Cited on page 10)

[84] *The X10 Programming Language*. http://www.x10-lang.org. (Cited on page 1)

[85] Li Yi, Christopher Moretti, Scott Emrich, Kenneth Judd, and Douglas Thain. Harnessing parallelism in multicore clusters with the all-pairs and wavefront abstractions. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 1–10, New York, NY, USA, 2009. ACM. (Cited on page 140)