Universidad Carlos III de Madrid

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

# PATRONES DE PARALELISMO: UNA APROXIMACIÓN BASADA EN BIBILIOTECAS GÉNERICAS

**Autor**: Víctor González Sánchez

**Tutor**: José Daniel García Sánchez

# Agradecimientos

Mis más sinceros agradecimientos a:

Mi tutor José Daniel García por haber estado tan atento a mi trabajo y haberme dado su apoyo y depositar su confianza en mí.

A todos los profesores, con lo que he tenido el placer de coincidir, por todos los conocimientos que me han proporcionado.

A mi familia por su confianza, apoyo y animo que me han dado en todo momento.

A mis compañeros de clase por su apoyo y ayuda durante el proyecto y toda la carrera.

A mis amigos y a mi novia por tranquilizarme y ayudarme a desconectar cuando el estrés podía conmigo.

# Índice general

# Ilustraciones

# Tablas

# 1. Introducción

Este primer apartado expone la visión general de este Trabajo Fin de grado, la motivación por la que ha sido realizado, los objetivos que tenía en su inicio, la estructura del documento y un glosario de términos para que la lectura sea más sencilla.

## 1.1 Motivación

Durante los últimos años uno de los mayores objetivos a conseguir por los ingenieros ha sido el aumento del rendimiento en los procesadores. Primeramente se centraron en explotar la frecuencia de reloj de los procesadores para mejorar el rendimiento de estos. Para conseguir esto lo que se hace es aumentar el número de transistores, en el año 1965 Moore enunció su ley, la cual fue rectificada en el año 1975, en dicha ley se afirmó que el número de transistores de los procesadores aumentarían duplicando su número cada 24 meses. Dicho aumento se ha cumplido hasta una fecha bastante reciente gracias a la reducción del tamaño de las puertas lógicas, pero este aumento de los transistores se ha visto frenado a causa de los límites físicos de los semiconductores utilizados, esto ha ocasionado problemas relacionados con el consumo, la disipación del calor y la necesidad de sincronizar la información. Con objetivo de lograr un mejor rendimiento en estos procesadores que eran secuenciales, se introdujeron instrucciones vectoriales que utilizaban el modelo SIMD (Single Intruction Multiple Data). Gracias a esto se conseguía el paralelismo a nivel de datos, esto consistía en aplicar la operación de una instrucción sobre un gran conjunto de datos. Esta mejora no fue suficiente y la viabilidad de esta arquitectura tuvo que ser reemplazada por nuevas generaciones de procesadores.

Como se observa en la Ilustración 1, se puede ver como hace ya más de una década se llegó al límite de la frecuencia de reloj de los procesadores, el consumo y la paralelización a nivel de instrucción. Con esto llego la revolución que ha hecho llegar a los llamados “multi-cores”, estos procesadores combinan varios procesadores en un mismo circuito integrado. Gracias a esto se puede conseguir que varios programas se estén ejecutando en diferentes núcleos o lo que es mejor que un mismo programa este siendo ejecutado en varios núcleos. Para hacer uso de esta arquitectura se explota el paralelismo a nivel de hilo, esto son procesos ligeros que pueden ser planificados por el sistema operativo. Los hilos son creados por un proceso padre que puede crear tantos hilos como soporte el multi-core, de esta forma se pueden ejecutar concurrentemente diferentes líneas de código de un mismo programa.

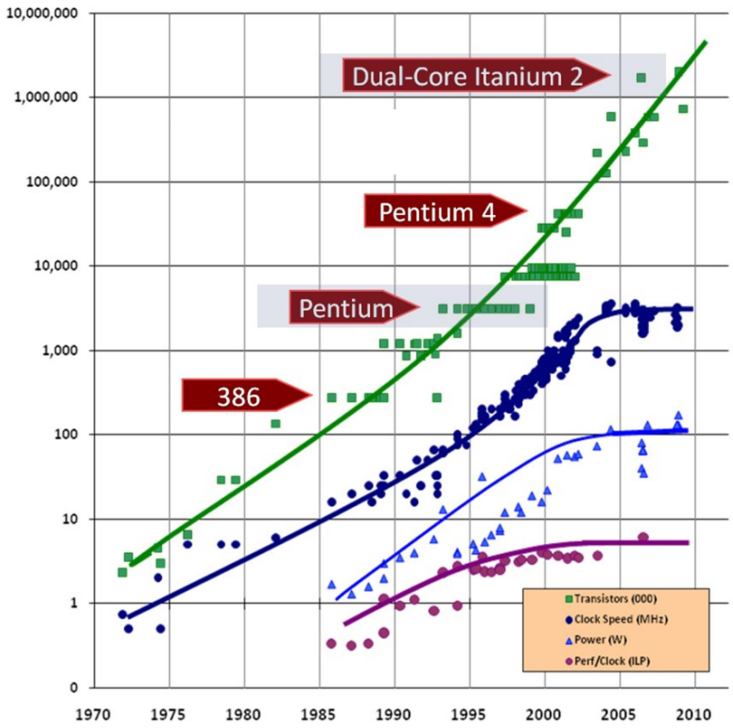



Ilustración 1: Tendencia de los procesadores según la ley de Moore

Fuente: A Fundamental Turn Toward Concurrency in Software [1]

La programación concurrente es bastante complicada y tediosa, es cierto que la aparición de bibliotecas de funciones que extienden de un lenguaje de secuencia como POSIX Threads o ISO Threads, o de lenguajes paralelos, como OpenMP o TBB han facilitado la programación paralela, pero no evitan que haya que preocuparse de problemas de escalabilidad, sincronización entre hilos o bloqueos generados.

Dado esta complejidad a la hora de programar de forma paralela surge la idea de crear una biblioteca paralela de algoritmos sencillos que son utilizados regularmente en la programación. Estos algoritmos son los que componen la STL de C++. Dichos algoritmos pueden ser utilizados de forma sencilla para cada uno de los contenedores que proporción la STL, estos son por ejemplo vectores, mapas , listas…, pero no solo para dichos contenedores sino también para contenedores creados por el usuario que cumplan la estructura de la STL. Esto es posible porque la programación de estos es genérica, es decir, es independiente del tipo de dato. Con dicha biblioteca el usuario se vería abstraído de los problemas de la paralelización y podría hacer uso de dichos algoritmos como si de forma secuencial se tratase y además con varias tecnologías de paralelización.

A todo lo descrito anteriormente, he de destacar mi interés personal por hacer un proyecto en C++, debido a que es uno de los lenguajes más utilizados en la programación y que se utiliza muy poco a lo largo de la carrera. Además del interés por conocer varias técnicas de paralelismo que pueden ser utilizadas, que tampoco es un aspecto en el que se profundice mucho a lo largo de la carrera.

## 1.2 Objetivos

El objetivo principal de este Trabajo Fin de Grado es la creación de una biblioteca que contenga los algoritmos de la STL de C++ paralelizados, dado el gran número de ellos se procederá a la paralelización de un numero representativo para demostrar la viabilidad de la paralelización de todos. Dichos algoritmos serán paralelizados con tres tecnologías: ISO Threads, OpenMP y TBB. La biblioteca también dará facilidad de cambiar de una tecnología a otra de forma sencilla. También este Trabajo Fin de Grado realizará un análisis de los speedups (aceleraciones) conseguidos de esta versión de los algoritmos respecto a su versión secuencial. Así mismo, se plantea una serie de objetivos secundarios:

- Evaluar las distintas aplicaciones de la suite del benchmark PARSEC con el fin de descubrir la utilidad de un benchmark y escoger una aplicación real en la que se pueda hacer uso de nuestra biblioteca.
- Modernizar el código de la aplicación elegida del benchmark PARSEC, en nuestro caso pasar un código puro de C a C++.
- Creación de pequeños benchmarks para realizar las pruebas de cada uno de los algoritmos.
- Mostrar una visión global del paralelismo en arquitecturas de memoria compartida, haciendo especial hincAPIé en las tecnologías ISO Threads, OpenMP y TBB.
- Comprobar si hay grandes diferencias de rendimiento entre OpenMP, TBB o el uso de ISO Threads.

## 1.3 Estructura del documento

Dentro de este punto, se listan los distintos apartados que componen el documento, así como una descripción del contenido:

- Apartado 1, Introducción: Determina la motivación por la que se ha realizado este Trabajo Fin de Grado, los objetivos que se pretenden conseguir con la consecución del mismo y una lista con los acrónimos principales.
- Apartado 2, Estado del Arte: Describe la base teórica sobre la que se sustenta el trabajo realizado, haciendo empeño en las tecnologías y aplicaciones utilizadas.
- Apartado 3, Planteamiento del problema: Lista y detalla los requisitos del trabajo a realizar y las restricciones en el diseño de la solución y su implementación.
- Apartado 4, Diseño e implementación: Describe el proceso realizado hasta llegar a los objetivos planteados y cumplir los requisitos impuestos.
- Apartado 5, Caso práctico: Blackscholes: Explica la modernización del código realizada y su preparación para el uso de la biblioteca.
- Apartado 6, Evaluación de resultados: Específica el hardware utilizado, la metodología para la medición de tiempos y los resultados finales.
- Apartado 7, Planificación del trabajo: Detalla las tareas realizadas y la estimación de tiempo que predijo, además las horas reales que costaron las tareas y un diagrama Trabajo Fin de Grado con las fechas en las que se realizaron.
- Apartado 8, Presupuesto: Especifica el cálculo del coste global de la realización del Trabajo Fin de Grado.
- Apartado 9, Marco legal y normativo: Especifica los estándares en los que está basado y las licencias que tiene relacionadas.
- Apartado 10, Conclusiones: Contiene un resumen que muestra el cumplimiento de los requisitos, las posibles líneas de investigación posteriores a este trabajo, unas conclusiones técnicas y una opinión personal del trabajo realizado.
- Apartado 12, Summary in English: Contiene un resumen en inglés.
- Apartado 11, Bibliografía: Contiene todas las referencias bibliográficas.

## 1.4 Acrónimos

| Termino | Descripción |
|---|---|
| **API** | Application Programming Interface |
| **ArBB** | Array Building Blocks |
| **CUDA** | Compute Unified Device Architecture |
| **DLP** | Data-Level Parallelism |
| **ILP** | Instruction-Level Parallelism |
| **ISO Threads** | Threads of International Organization for Standardization |
| **MIT** | Massachusetts Institute of Technology |
| **NPTL** | Native POSIX Thread Library |
| **NUMA** | Non-Uniform Memory Access |
| **OpenMP** | Open Multiprocessing |
| **PARSEC** | Princeton Application Repository for Shared-Memory Computers |
| **Pthreads** | POSIX Threads |
| **RAM** | Random access memory |
| **SIMD** | Single Instruction Multiple Data |
| **STL** | Standard Template Library |
| **SO** | System operative |
| **TBB** | Threading Building Blocks |
| **TLP** | Thread-Level Parallelism |

Tabla 1: Acrónimos

# 2. Estado del arte

En este segundo apartado, se describe la base teórica del Trabajo Fin de Grado. En este punto es necesario describir el paralelismo en arquitecturas de memoria compartida, algoritmos pertenecientes a la STL y las tecnologías posibles y utilizadas para realizar los algoritmos paralelos. Además de aportar una idea general de que es un benchmark, explicar el contenido del Benchmark PARSEC y explicar detalladamente en que consiste la aplicación elegida de PARSEC, donde se analizarán los resultados de un algoritmo.

## 2.1 Modelos de paralelismo y arquitecturas de memoria compartida

El paralelismo [2] es un tipo de computación en el cual muchos cálculos son realizados simultáneamente, actuando sobre el principio de que los problemas grandes a menudo pueden estar divididos en más pequeños, en el que dichos problemas pequeños son solucionados al mismo tiempo. Gracias a esta técnica, se consigue aumentar el rendimiento de las aplicaciones, reduciendo su tiempo de cómputo. Cuando se habla de paralelismo se puede hablar de varios tipos:

### *Paralelismo a nivel de bits*

Este paralelismo está centrado en aumentar el ancho de palabra, con esto lo que se consigue es reducir el número de instrucciones necesarias para realizar una operación en variables cuyo tamaño es mayor que el tamaño de palabra. Un ejemplo sencillo es cuando un computador de 8 bits debe sumar dos enteros de 16 bits, el procesador primero tiene que adicionar los 8 bits de orden inferior a través de la instrucción de adicción de los dos enteros y después realizar lo mismo con los 8 bits de orden superior con la de adicción con acarreo que tiene en cuenta los bits de nivel inferior. Este procesador necesita dos instrucciones para realizar la adicción pero en cambio uno de 16 bits solo necesitaría una.

### *Paralelismo a nivel de instrucción (ILP)*

Este paralelismo se centra en conseguir ejecutar varias instrucciones al mismo tiempo en un mismo procesador. Esto es posible ya que un programa es una secuencia de instrucciones a ejecutar por el procesador, dichas instrucciones tiene varias etapas (pipeline) que son acciones diferentes que tiene que hacer el procesador sobre una instrucción. Por lo tanto el procesador podría realizar la acción de cada etapa simultáneamente, y dichas etapas fuesen de instrucciones diferentes. Con esto se consigue que el procesador este ejecutando varias instrucciones que estén por etapas diferentes. Cuanto mayor sea el número de etapas más paralelización de instrucciones se puede realizar, por ejemplo el Pentium 4 tenía una pipeline de 35 etapas.

### *Paralelismo a nivel de Datos (DLP)*

Este paralelismo se centra en dividir un conjunto de datos de un programa en subconjuntos que son asignados a cada uno de los núcleos de la computadora. Cada núcleo realizara la misma secuencia de operaciones que los otros núcleos pero sobre datos diferentes. En conclusión redistribuyen los datos y se replican las tareas. Este paralelismo fue conseguido gracias a la arquitectura SIMD (Ilustración 2) que propuso Flynn en el año 1972. Como fue citado anteriormente la arquitectura SIMD también permite realizar una única operación sobre varios datos a la vez, esto es posible si el procesador posee unidades vectoriales.

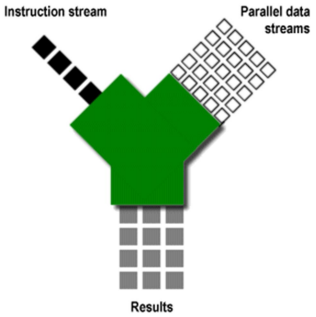



**Ilustración 2: Arquitectura SIMD**

**Fuente: http://arstechnica.com/features/2000/03/simd/**

### *Paralelismo a nivel de Tareas (TLP)*

Este paralelismo se centra en asignar distintas secuencias de operaciones a cada núcleo de la computadora. Para esto se usan los llamados hilos o threads de lo que ya se ha hablado anteriormente. Cada uno sería un proceso que ejecutaría la tarea que se le haya sido asignada.

Una vez conociendo todos los modelos de paralelismo, nos centraremos un poco más en el último del que hemos hablado, ya que es el que se ha utilizado para elaborar la biblioteca de algoritmos paralela.

Centrándonos en los hilos que se pueden ejecutar al mismo tiempo, esto está muy relacionado con el número de núcleos del computador. En cada uno de los núcleos se ejecutaría un hilo diferente, pero esto no significa que si nuestra computadora tiene dos núcleos solo puedan crearse dos hilos, el número de hilos puede ser mucho mayor y estos serán planificados por el SO. Por lo que no hay que pensar que cuanto mayor número de hilos más veloz va a ser nuestro programa, debido a que la planificación y creación de dicho hilos también tienen un tiempo de cómputo. También hay que añadir que no mayor número de núcleos se traduce en mayor rendimiento, también hay que tener en cuenta la frecuencia de los núcleos ya que de esta dependerá la cantidad de instrucciones que se pueden ejecutar por unidad de tiempo.

En los últimos años se ha desarrollado una técnica llamada multi-hilo como es la conocida hyper-threading de los procesadores Intel. Con esto se consigue que en un mismo núcleo se estén ejecutando varios hilos a la vez. Estos son los llamados núcleos lógicos. Por ejemplo un computador que tenga 4 núcleos físicos gracias al hyper-threading podría tener 8 núcleos lógicos, lo que viene a ser ejecutar 8 hilos al mismo tiempo.

Una vez que conocemos la paralelización y como es el uso de varios núcleos en un computador, vamos a ver cómo se gestiona la memoria del computador algo esencial ya que es uno de los mayores problemas al utilizar varios núcleos. Esto son las llamadas arquitecturas de memoria compartida donde todos los núcleos comparten el mismo rango de direcciones de memoria. Dentro de los multiprocesadores aparecen dos tipos de diseños para el acceso a memoria [3].

**Arquitectura de memoria de acceso uniforme (UMA) (Ilustración 3)**

Este diseño permite que todos los procesadores puedan acceder a las distintas posiciones de memoria invirtiendo el mismo tiempo de acceso. Cada uno de los procesadores puede tener una cache privada.

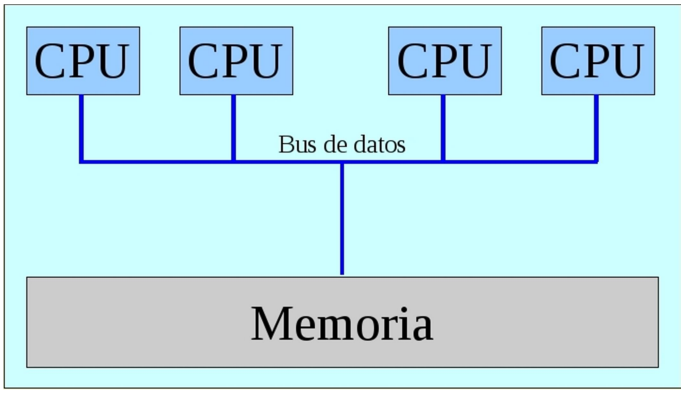



Ilustración 3: Arquitectura UMA

Fuente: http://www.muylinux.com/2008/05/21/numa-es-gran-desconocido

**Arquitectura de memoria de acceso no uniforme (NUMA) (Ilustración 4)**

En esta arquitectura el tiempo de acceso a un dato depende del núcleo que solicite la petición y donde este dicho dato. Un núcleo tardará más tiempo en acceder a un dato que esté en la memoria de otro procesador que a un dato que esté en la suya. La principal ventaja de esta arquitectura es la escalabilidad ya que permite añadir nuevos nodos sin mucho problema. La principal desventaja es que al tener bancos de memoria independientes, se necesita de un sistema que se encargue de la sincronización y coherencia de datos en cada una de las memorias, lo que ocasiona que los fallos de cache de un proceso ejecutando en varios procesadores al mismo tiempo añada una

gran penalización en el tiempo de cómputo. Como en el caso de UMA todos los núcleos pueden tener una cache privada.

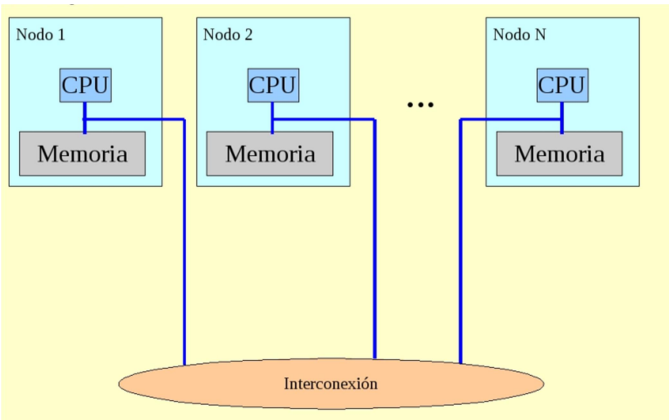



**Ilustración 4: Arquitectura NUMA**

**Fuente: http://www.muylinux.com/2008/05/21/numa-es-gran-desconocido**

La memoria cache de los núcleos en ambas arquitecturas nos permite reducir considerablemente el tiempo de acceso a los datos, dependiendo del tamaño de cache y conociendo los principios de localidad temporal y espacial es posible reducir el número de fallos de cache y en consecuencia el tiempo de ejecución del programa.

Una vez conocida toda esta información queda a cargo del desarrollador conocer las arquitecturas en las que va a desarrollar y en las que se va a utilizar dicho programa para averiguar si cierto código puede ser paralelizado en ciertas computadoras. Esto no es una tarea sencilla ya que depende de muchos factores como tipos de variables que se utilizan, operaciones aritméticas o de coma flotante, numero de procesadores, accesos a memoria, etc.

La última tarea de la paralelización es análisis de los resultados comparando los tiempos de la versión secuencial del programa con los de la versión paralela. Uno de los primero factores a calcular es la aceleración global (speedup) a través de la siguiente formula:

$$Aceleración = \frac{Tiempo\ en\ version\ secuencial}{Tiempo\ en\ version\ paralela}$$

Este resultado muestra el número de veces que es más rápida la versión paralela respecto la secuencial. Esto también se puede utilizar para comparar distintas versiones paralelas. Sin embargo, este resultado no tiene en cuenta el número de núcleos que se han utilizado, información importante si queremos saber que aceleración es la que se consigue por cada procesador, para esto se puede utilizar la siguiente formula:

$$Aceleración\ por\ procesador = \frac{Aceleración}{Numero\ de\ procesadores}$$

Es muy importante tener en cuenta cual es la mayor aceleración que es posible obtener de un código, esto se puede calcular gracias a la ley de Amdahl. Para ello es necesario conocer la parte secuencial que no se modificó y el número de núcleos que dispone el entorno de ejecución. La fórmula es la siguiente:

$$Aceleración\ máxima = \frac{1}{(1-P) + \frac{P}{N}}$$

Donde P es la proporción de tiempo que ocupa la parte paralelizada y N el número de procesadores.

## 2.2 Biblioteca STL y sus algoritmos

La STL [4] es una biblioteca diseñada por Alex Stepanov con objetivo de ser lo más general y eficiente posible. Esta biblioteca está compuesta por tres elementos (Ilustración 5), contenedores como pueden ser vectores, listas, mapas o árboles, algoritmos que son funciones que manipulan los datos pero sin tener conocimiento de los contenedores y la que relaciona estas dos que son los iteradores. Cada contenedor tiene un tipo propio de iterador y estos son los que son utilizados por los algoritmos para manipular los datos.

Esta biblioteca es genérica por lo tanto todo contenedor puede utilizarse con todo tipo de datos, tanto tipos básicos, como clases y estructuras creados por el usuario. Además también permite crear tus propios contenedores e iteradores para utilizar los algoritmos.

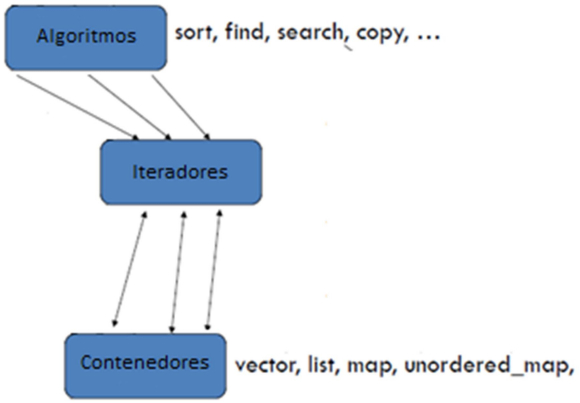



Ilustración 5: Componentes de la STL

Esta biblioteca incluye más de diez contenedores implementados, estos son la base de la programación y sin ninguna duda en casi todo código de programación aparece alguno de estos, de ahí su importancia. Por otra parte los algoritmos facilitan mucho su uso, la biblioteca dispone de más de ochenta algoritmos y además con la ventaja de que dichos algoritmos están muy optimizados.

A continuación se muestra un listado de los algoritmos clasificados por familias y una breve descripción de lo que realizan:

| Algoritmos que no modifican la secuencia | |
|---|---|
| **all_of**<br>**any_of**<br>**none_of** | Comprueba si un predicado es verdadero para todos, cualquiera o ninguno de los elementos de un intervalo |
| **for_each** | Se aplica una función a una serie de elementos |
| **count**<br>**count_if** | Devuelve el número de elementos que cumplan criterios específicos |
| **mismatch** | Encuentra la primera posición donde dos rangos difieren |
| **equal** | Determina si dos conjuntos de elementos son los mismos |
| **find**<br>**find_if**<br>**find_if_not** | Encuentra el primer elemento que satisfaga los criterios específicos |
| **find_end** | Encuentra la última secuencia de elementos en un cierto rango |
| **find_first_of** | Busca uno cualquiera del conjunto de elementos. |
| **adjacent_find** | Encuentra dos elementos idénticos (o alguna otra relación) adyacentes el uno al otro |
| **search** | Busca en un rango de elementos |
| **search_n** | Busca un número de copias consecutivas de un elemento en un rango |

Tabla 2: Algoritmos que no modifican la secuencia

| Algoritmos que modifican la secuencia | |
|---|---|
| **copy**<br>**copy_if** | Copia un rango de elementos a una nueva ubicación |
| **copy_n** | Copia un número de elementos a una nueva ubicación |
| **copy_backward** | Copia un intervalo de elementos en orden inverso |
| **move** | Mueve una serie de elementos a una nueva ubicación |
| **move_backward** | Mueve una serie de elementos a una nueva ubicación en orden inverso |
| **fill** | Asigna a una serie de elementos un cierto valor |
| **fill_n** | Asigna un valor a un número de elementos |
| **transform** | Se aplica una función a una serie de elementos |
| **generate** | Guarda el resultado de una función en un intervalo |
| **generate_n** | Guarda el resultado de realizar la función a N elementos |
| **remove**<br>**remove_if** | Elimina elementos que cumplan criterios específicos |
| **remove_copy**<br>**remove_copy_if** | Copia un intervalo de elementos omitiendo los que satisfacen criterios específicos |
| **replace**<br>**replace_if** | Sustituye a todos los valores que satisfacen los criterios específicos con otro valor |
| **replace_copy**<br>**replace_copy_if** | Copia un intervalo, en sustitución de elementos que cumplan criterios específicos con otro valor |
| **swap** | Intercambia los valores de dos objetos |
| **swap_ranges** | Intercambia dos rangos de elementos |
| **iter_swap** | Intercambia los elementos apuntado por dos iteradores |
| **reverse** | Invierte los elementos de orden en un intervalo |
| **reverse_copy** | Crea una copia de un rango que se invierte |
| **rotate** | Gira el orden de los elementos en un rango |
| **rotate_copy** | Copia y gira una serie de elementos |
| **random_shuffle**<br>**shuffle** | El azar reordena los elementos de un rango |
| **unique** | Remueve elementos duplicados consecutivos en un rango. |
| **unique_copy** | Crea una copia de un cierto rango de elementos que no contiene duplicados consecutivos. |

Tabla 3: Algoritmos que modifican la secuencia

| Algoritmos de particionamiento | |
|---|---|
| **is_partitioned** | Determina si el intervalo está dividido por el predicado dado |
| **partition** | Divide una serie de elementos en dos grupos |
| **partition_copy** | Copia un rango dividido en dos grupos |
| **stable_partition** | Divide en dos grupos, preservando su orden relativo |
| **partition_point** | Localiza el punto de partición |

Tabla 4: Algoritmos de particionamiento

| Algoritmos de clasificación | |
|---|---|
| **is_sorted** | Comprueba si un rango se clasifican en orden ascendente |
| **is_sorted_until** | Encuentra el mayor subrango ordenados |
| **sort** | Ordena un intervalo en orden ascendente |
| **partial_sort** | Ordena los primeros n elementos de un rango |
| **partial_sort_copy** | Copias y clasifica parcialmente una serie de elementos |
| **stable_sort** | Ordena un intervalo de elementos, mientras que la preservación del orden entre los elementos iguales |
| **stable_sort** | Ordena un intervalo de elementos, mientras que la preservación del orden entre los elementos iguales |
| **nth_element** | Parcialmente ordena el rango dado asegurándose de que está dividida por el elemento dado |

Tabla 5: Algoritmos de clasificación

| Algoritmos de búsqueda binaria (en rangos ordenados) | |
|---|---|
| **lower_bound** | Devuelve un iterador que contiene el primer elemento que no es menor que el valor dado. |
| **upper_bound** | Devuelve un iterador que contiene el primer elemento que no es mayor que el valor dado. |
| **binary_search** | Determina si existe un elemento en un cierto rango |
| **equal_range** | Devuelve un rango de elementos que coinciden con una clave específica |

Tabla 6: Algoritmos de búsqueda binaria

| Algoritmos de conjunto (en rangos ordenados) | |
|---|---|
| **merge** | Fusiona dos rangos ordenados |
| **inplace_merge** | Fusiona dos rangos ordenados en el lugar |
| **includes** | Devuelve verdadero si un grupo es un subconjunto de otro |
| **set_difference** | Calcula la diferencia entre los dos conjuntos |
| **set_difference** | Calcula la diferencia entre los dos conjuntos |
| **set_intersection** | Calcula la intersección de dos conjuntos |
| **set_symmetric_difference** | Calcula la diferencia simétrica entre dos conjuntos |
| **set_union** | Calcula la unión de dos conjuntos |

Tabla 7: Algoritmos de conjunto

| Algoritmos de montículo | |
|---|---|
| **is_heap** | Comprueba si el rango dando es el máximo del montículo |
| **is_heap_until** | Encuentra el mayor subrango que sea el máximo del montículo |
| **make_heap** | Crea un montículo máximo de una serie de elementos |
| **push_heap** | Añade un elemento a un máximo montículo. |
| **pop_heap** | Elimina el elemento más grande de máximo montículo |
| **sort_heap** | Cambia el máximo montículo en un rango de elementos ordenados ascendentemente. |

Tabla 8: Algoritmos de montículo

| Algoritmos de mínimo y máximo | |
|---|---|
| **max** | Devuelve el mayor de los dos elementos |
| **max_element** | Devuelve el elemento más grande de un rango |
| **min** | Devuelve el menor de los dos elementos |
| **min_element** | Devuelve el elemento más pequeño de un rango |
| **minmax** | Devuelve el elemento más grande y el más pequeño de los dos elementos |
| **minmax_element** | Devuelve el elemento más pequeño y el más grande en un rango |
| **lexicographical_compare** | Devuelve verdadero si el rango es menor que otro lexicográficamente hablando. |
| **is_permutation** | Determina si una secuencia de elementos es una permutación u otra secuencia. |
| **next_permutation** | Genera la siguiente mayor lexicográfica permutación de un rango de elementos. |
| **prev_permutation** | Genera la siguiente menor lexicográfica permutación de un rango de elementos. |

Tabla 9: Algoritmos de mínimo y máximo

| Algoritmos numéricos | |
|---|---|
| **iota** | Llena un rango con incrementos sucesivos del mismo valor de partida |
| **accumulate** | Suma una serie de elementos |
| **inner_product** | Calcula el producto interno de dos rangos de elementos |
| **adjacent_difference** | Calcula las diferencias entre elementos adyacentes en un rango |
| **partial_sum** | Calcula la suma parcial de una serie de elementos |

Tabla 10: Algoritmos numéricos

## 2.3 Mecanismos de concurrencia

La concurrencia es la capacidad de ejecutar varias tareas al mismo tiempo, pero gracias a esta concurrencia se puede conseguir paralelismo. Esto se consigue al dividir una misma tarea en tareas más pequeñas y ejecutarlas concurrentemente. A continuación se explica dos mecanismos de concurrencia que pueden ser usados para conseguir paralelismo.

### 2.3.1 POSIX Threads

POSIX Threads (Pthreads) [5] es la especificación de una API que nos permite la programación concurrente en sistemas con arquitectura multi-core y memoria compartida. Fue definida en el estándar ANSI/IEE POSIX 1003.1c [6] en 1995. En este momento la versión más popular y extendida es la llamada NPTL (Native Posix Thread Library) que fue incluida en el kernel de linux en 2003.

Esta API hace uso de los denominados threads para proporcionar paralelismo a nivel de hilo. Estos a diferencia de un proceso que es un programa en ejecución independiente del resto de procesos, un hilo es un subproceso que comparte memoria con otros hilos y el proceso principal. Sin embargo, todo hilo tiene su propio contador de programa, registros y estado. Al compartir información, es necesario añadir mecanismos de sincronización con el objetivo de evitar condiciones de carrera. Esto hace que la programación sea más compleja ya que hay que ocuparse de dicha sincronización.

Relacionado con los hilos existen dos tipos, a nivel de usuario, donde la creación y gestión de los hilos debe ser gestionada por la aplicación y los hilos a nivel de kernel, donde el kernel planifica múltiples hilos del mismo proceso en múltiples procesadores. NPTL soporta ambos tipos, pero los hilos de nivel usuario no tienen alcance en la programación concurrente en arquitecturas de multiprocesadores debido a que la creación de hilos se realiza en un único procesador.

Las subrutinas que comprenden el API se pueden agrupar en cuatro grupos:

- Gestión de hilos: Son las rutinas que permiten trabajar directamente con los hilos, creándolos, esperando por ellos, separar el padre de los hilos hijos o por ejemplo consultar información de un hilo.
- Mutex: Son la rutinas que tienen que ver con la sincronización, se llaman mutex ya que son una abreviatura de *mutual exclusion.* Las funciones de exclusión mutua permiten crear, destruir, bloquear y desbloquear los mutex.
- Variables de condición: Rutinas que se ocupan de la comunicación entre hilos que comparten un mutex. Este grupo contiene funciones para crear, destruir y la señal de esperar sobre una base de valores de las variables especificadas.
- Sincronización: Rutinas que gestionan los bloqueos de lectura, escritura y barreras.

La API Pthreads nos permite un gran control en la gestión de hilos y la capacidad de aprovechar las especificaciones de cada computador, se puede decir que nos acerca al hardware a cambio de tener problemas con la sincronización de la información y siendo más laboriosa la programación.

Un ejemplo sencillo de cómo se lanza un hilo y se esperaría por él sería:

```
void *func(void* a){
      int aux=*(int *)a;
      do_something(aux);
}
int main(){
      void *func();
      pthread_t  thid;
      int a=4;
      pthread_create(&thid, NULL, func,&a);
      pthread_join(thid, NULL);
      return 0;
}
```

### 2.3.2 ISO Threads en C++11 y C11

ISO Threads es la especificación de un API que nos permite programación concurrente en sistemas con arquitectura multi-core y memoria compartida. Fue definida en el estándar ISO/IEC 14882:2011 [7] para el lenguaje C++ y en el estándar ISO/IEC 9899:2011 [8] para el lenguaje C, después fue revisado para el lenguaje C++ y fue especificado en el estándar ISO/IEC 14882:2014 [9].

Centrándonos en ISO Threads C++11[10] podríamos decir que es muy parecido a Pthreads pero dándonos más portabilidad, debido a que este estándar funciona para varios tipos de hilos, como pueden ser los hilos de POSIX o los de Windows. También el uso es más sencillo que Pthreads debido a que no tiene uso de punteros. De esta forma es posible portar programas de un SO a otro sin mucho problema.

Las subrutinas que comprenden el API dan soporte a:

- Gestión de hilos: Son las subrutinas que nos permiten trabajar con los hilos como puede ser crearlos, esperar por ellos, obtener el id del hilo que se está ejecutando, etc.
- Exclusión mutua: Estas subrutinas permiten a varios hilos protegerse del acceso simultáneo a recursos compartidos. Esto permite evitar tener datos corruptos y proporciona las bases para la sincronización de hilos.
- Variables de condición: Son objetos de sincronización que permite que varios hilos se comuniquen. Permite por ejemplo que un hilo espere hasta que otro hilo realice otra acción.
- Futuros: El estándar proporciona facilidades para la obtención de los valores que se devuelven y para recuperar las excepciones producidas por tareas asíncronas (es decir, las funciones puestas en marcha por hilos). Estos valores están en un "estado compartido", y de forma asíncrona se le puede poner un retorno o una excepción que será examinada por otros hilos.

Un ejemplo sencillo de cómo se lanza un hilo y se esperaría por él sería:

```
void func(int a){
      do_something(a);
}
int main(){
      int a=5;
      std::thread first (func,a);
      first.join();
      return 0;
}
```

## 2.4 Mecanismos de paralelismo

### 2.4.1 OpenMP

El estándar Open Multi-Processing (OpenMP) [11], fue lanzado en 1997, este estándar especifica un API que utiliza el paralelismo a nivel de hilos para arquitecturas con memoria compartida. OpenMP utiliza directivas del compilador y llamadas a subrutinas para realizar una paralelización del código. OpenMP es soportado solo por los lenguajes C, C++ y Fortran. Como OpenMP está basado en directivas que se hacen referencia dentro del código secuencial, se evita tener que realizar grandes modificaciones del código para conseguir paralelizar este.

Este API utiliza una estrategia en la que se comienza con un único hilo llamado hilo maestro, cuando llega a una directiva de OpenMP, la carga de trabajo se divide para que la sección de la directiva sea ejecutada por varios hilos. Cuando la sección a paralelizar termina se destruyen todos los hilos quedando el hilo maestro.

En los comienzos de OpenMP estaba orientado solamente en la paralelización de grandes bucles que fuesen regulares, sin embargo, la nueva versión ha introducido un modo de paralelización por tareas. Con este modelo no se puede tener un control sobre el hardware como con Pthreads o ISO Threads, pero si evita tener que hacerse cargo de las sincronizaciones de variables de una forma rudimentaria, ya que nos permite indicar por ejemplo variables privadas o compartidas o secciones críticas.

Una de las desventajas de este modelo es que no incorpora funcionalidad para realizar la paralelización de hilos en sistemas distribuidos y que no es muy eficiente en arquitecturas NUMA, ya que no se puede explotar el principio de localidad de los datos ni establecer afinidad entre los hilos y los núcleos.

Un ejemplo sencillo de uso para paralelizar un bucle sería el siguiente:

```
int array[1000];
#pragma omp parallel for
for(int i=0, i<1000, i++){
array[i]=func();
}
```

### 2.4.2 Intel Threading Building Blocks

Intel Threading building Blocks (TBB) [12] tuvo su primera versión en el año 2006 con el objetivo de ofrecer un alto nivel de abstracción en la paralelización. Este modelo se basa en la paralelización basada en tareas y es soportado por Windows, Linux y MAC.

Además tiene la ventaja que es compatible con otros modelos como el ya citado OpenMP o por ejemplo ArBB.

TBB utiliza un modelo de tareas, en el que divide el trabajo en pequeñas tareas a diferencia de Pthreads, ISO Threads y OpenMP que lo hacían por medio de hilos, aunque es cierto que OpenMP también añadió un modelo de tareas en sus últimas versiones. Dichas tareas son asignadas a los hilos de forma automática.

Existen varias ventajas de utilizar tareas en vez de hilos a continuación están expuestas las más importantes:

- TBB incorpora una técnica en la que si un núcleo ha terminado con su cola de tareas y va a quedar libre se le asignan automáticamente tareas de otro núcleo que todavía tenga tareas en su cola. Esto es muy beneficioso ya que evita que un núcleo se quede en estado pasivo.
- TBB crea todos los hilos disponibles al principio y solo los destruye al final de programa no como realiza OpenMP. TBB solo realiza operaciones de creación y destrucción tareas continuamente que es menos costoso que realizarlo de hilos.
- TBB obtiene información que contiene la memoria cache para seleccionar el orden de ejecución más eficiente. Esta medida supone un gran aumento de rendimiento en casos que el tiempo de penalización de un fallo cache es muy grande.
- Como ocurría con OpenMP que mejoraba el nivel de abstracción de Pthreads e ISO Threads, en este caso TBB lo aumenta en cierta medida respecto a OpenMP gracias al uso de tareas

Hay que añadir que estas mejoras respecto OpenMP han disminuido ya que en OpenMP se ha introducido un modelo de tareas, pero que no es tan usado como en el caso de TBB. Esto es motivo de que con OpenMP se utiliza en gran medida el modelo antiguo debido a su simplicidad de paralelizar un bucle.

Un ejemplo sencillo de uso en el que se paraleliza un bucle sería el siguiente:

```
int array[1000];
TBB:parallel_for(0,1000,1,[=](int i)){
array[i]=func();
});
```

### 2.4.3 Algoritmos paralelos de C++17

El estándar ISO/IEC JTC1/SC22 [13] describe los requisitos para las implementaciones de una interfaz que los programas escritos en C++ pueden utilizar para invocar algoritmos con su ejecución en paralelo.

Dichas interfaces proporciona que el usuario elija el modo de ejecución por medio de políticas. Estas políticas son:

- sequential_execution_policy: La ejecución del algoritmo sería secuencial como son ahora mismo los algoritmos de la STL.
- parallel_execution_policy: La ejecución del algoritmo se hará de forma paralela siendo responsabilidad del programador evitar el bloqueo mutuo.
- parallel_vector_execution_policy: La ejecución del algoritmo se hará de forma paralela y vectorizada siendo responsabilidad del programador evitar el bloqueo mutuo.

Un ejemplo de uso sería el siguiente:

```
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::par_vec, std::begin(a), std::end(a), [&](int)
{
  std::lock_guard<std::mutex> guard(m);
  ++x;
});
```

## 2.5 Benchmark PARSEC

Un benchmark es una aplicación que tiene como objetivo medir el rendimiento del computador o de algún elemento específico que contiene, como puede ser el procesador, la memoria RAM o la tarjeta gráfica. Estas aplicaciones suelen ser aplicaciones muy eficientes y controlables. En general los propósitos de un benchmark son comparar el rendimiento de dos o más sistemas para averiguar cuál es mejor, evaluar el comportamiento de distintos componentes y averiguar las limitaciones del computador con objetivo de mejorar su rendimiento.

La suite benchmark PARSEC (Priceton Apllocation Repository for Shared-Memory Computers) [14] está compuesta por trece programas que tienen como objetivo evaluar procesadores multi-core en arquitecturas con memoria compartida. Esta suite

fue publicada por primera vez en el año 2008 donde instituciones de gran calibre como Princeton, Intel o el MIT colaboraron. Todos estos programas han sido paralelizados previamente y por su innovación y diversidad hacen que esta suite sea una de las más completas en este sector.

En la siguiente tabla se muestra un resumen de las características de cada uno de los programas.

| Programa | Dominio | Paralelización | | Carga de trabajo | Uso de datos | |
|---|---|---|---|---|---|---|
| | | Modelo | Granularidad | | Compartición | Intercambio |
| **Blackscholes** | Análisis financiero | Paralelización de datos | Grueso | Pequeña | Baja | Bajo |
| **Bodytrack** | Inteligencia artificial | Paralelización de datos | Medio | Media | Alta | Medio |
| **Canneal** | Ingeniería | No estructurado | Fino | Ilimitado | Alta | Alto |
| **Dedup** | Almacenamiento corporativo | Flujo de datos | Medio | Ilimitado | Alta | Alto |
| **Facesim** | Animación | Paralelización de datos | Grueso | Grande | Baja | Medio |
| **Ferret** | Búsqueda de similitudes | Flujo de datos | Medio | Ilimitado | Alta | Alto |
| **Fludiminate** | Animación | Paralelización de datos | Fino | Grande | Baja | Medio |
| **Freqmine** | Minería de datos | Paralelización de datos | Medio | Ilimitado | Alta | Medio |
| **StreamCluster** | Minería de datos | Paralelización de datos | Medio | Media | Baja | Medio |
| **Swaptions** | Análisis financiero | Paralelización de datos | Grueso | Media | Baja | Bajo |
| **Vips** | Procesado multimedia | Paralelización de datos | Grueso | Media | Baja | Medio |
| **X264** | Procesado multimedia | Flujo de datos | Grueso | Media | Alta | Alto |

Tabla 11: Programas incluidos en la primera versión de PARSEC

PARSEC además de los programas nos proporciona seis entradas para cada uno de los programas, gracias a estos de forma sencilla se puede comprobar su funcionamiento, obtener resultados que sirven de toma de contacto y también una entrada para la evaluación completa. De forma más específica las entradas que ofrece son:

- Test: entrada de tamaño muy pequeño que nos sirve para probar la funcionalidad básica.
- Simdev: Entrada de tamaño pequeño que sirve para hace una aproximación al comportamiento real de la aplicación. Suele ser usado para tareas de desarrollo.
- Simsmall, simmedium, simlarge: Entradas de diferentes tamaños convenientes para evaluar el rendimiento en micro arquitecturas.
- Native: Entrad de gran tamaño con el que se puede hacer una simulación real.

En la siguiente tabla se muestra un resumen de las trece aplicaciones, donde se pueden ver datos de interés, posterior a ella se describe de forma general la funcionalidad de cada una de ellas [15]:

| Programa | Modelo de Paralelización | | | Portabilidad | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Compilador | | | SO | | | CPU | | |
| | Pthreads | OpenMP | Intel TBB | GCC 4.2 | GCC 4.3 | icc 10.1 | Linux | Solaris 10 | Windows | i386 | x86_64 | Sparc |
| **blackscholes** | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si |
| **bodytrack** | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si | Si |
| **canneal** | Si | No | No | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **dedup** | Si | No | No | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **facesim** | Si | No | No | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **ferret** | Si | No | No | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **fluidanimate** | Si | No | Si | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **freqmine** | No | Si | No | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **raytrace** | Si | No | No | Si | Si | Si | Si | Si | Si | Si | Si | Si |
| **streamcluster** | Si | No | Si | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **swaptions** | Si | No | Si | Si | Si | Si | Si | Si | No | Si | Si | Si |
| **vips** | Si | No | No | Si | Si | Si | Si | Si | Si | Si | Si | Si |
| **x264** | Si | No | No | Si | Si | Si | Si | Si | No | Si | Si | Si |

Tabla 12: Listado de características de las Aplicaciones PARSEC

**Blackscholes**

Será explicada detenidamente más adelante, debido a ser la aplicación sobre la que se ha desarrollado el caso práctico.

**Bodytrack**

Esta aplicación de visión por ordenador es una carga de trabajo de Intel RMS que sigue un cuerpo humano con múltiples cámaras a través de una secuencia de imágenes. Este benchmark se incluyó debido a la creciente importancia de los algoritmos de visión por ordenador en áreas tales como la vigilancia de vídeo, animación de personajes y las interfaces de ordenador.

**Canneal**

Este kernel fue desarrollado por la Universidad de Princeton. Utiliza un modelo de cache especifica (cache-aware) de recocido simulado para reducir al mínimo el coste de encaminamiento en el diseño de un chip.

**Dedup**

Este kernel fue desarrollado por la Universidad de Princeton. Se comprime un flujo de datos con una combinación de compresión global y local que se llama "*deduplicación*". El kernel utiliza un modelo de programación "*pipeline*" para imitar las implementaciones del mundo real. La razón de la inclusión de este kernel es que la deduplicación se ha convertido en un método convencional para sistemas de almacenamiento de copia de seguridad de nueva generación.

**Facesim**

Esta aplicación Intel RMS fue desarrollado originalmente por la Universidad de Stanford. Se calcula una animación visual realista de la cara modelada mediante la simulación de la física fundamental. La carga de trabajo se incluyó en el conjunto de pruebas, ya que un número cada vez mayor de animaciones emplean la simulación física para crear efectos más realistas.

**Ferret**

Esta aplicación se basa en el kit de herramientas Ferret que se utiliza para la búsqueda por similitud. Fue desarrollado por la Universidad de Princeton. La razón de la inclusión en el conjunto de pruebas es que representa los motores de búsqueda de próximas generaciones emergentes para tipos de datos de documentos no textuales. Ferret está paralelizado utilizando el modelo de tubería.

**Fluidanimate**

Esta aplicación Intel RMS utiliza una extensión del método de Hidrodinámica de Partículas Suavizadas (SPH) para simular un fluido incompresible para los propósitos de animación interactivas. Fue incluido en el conjunto de pruebas PARSEC debido a la creciente importancia de las simulaciones físicas para las animaciones.

**Freqmine**

Esta aplicación cuenta con una versión basada en matrices del método FP-growth (Patrón frecuente de crecimiento) y ejecuta el método de minería llamado conjunto de elementos frecuentes. Es un punto de referencia de Intel RMS que fue desarrollado originalmente por la Universidad de Concordia. Freqmine se incluyó en el conjunto de pruebas PARSEC debido a la creciente utilización de técnicas de minería de datos.

R**aytrace**

La aplicación Intel RMS utiliza una versión del método de trazado de rayos que normalmente se emplea para las animaciones en tiempo real, como en juegos de ordenador. Está optimizado para la velocidad más que para el realismo. La complejidad computacional del algoritmo depende de la resolución de la imagen de salida y la escena.

**Streamcluster**

Este núcleo RMS fue desarrollado por la Universidad de Princeton y resuelve el problema en línea del algoritmo de agrupamiento. Streamcluster se incluyó en el conjunto de pruebas PARSEC debido a la importancia de los algoritmos de minería de datos y la prevalencia de los problemas con las características de transmisión.

**Swaptions**

La aplicación es una carga de trabajo de Intel RMS que utiliza el framework Heath-Jarrow-Morton basado en el modelo de Monte Carlofijar para calcular los precios de un portafolio de opciones sobre cambios (MC) de simulación.

**Vips**

Esta aplicación se basa en el Sistema de Procesamiento de Imágenes VASARI (VIPS) que fue originalmente desarrollado a través de varios proyectos financiados por la Unión Europea (UE). El punto de referencia incluye operaciones de imagen fundamentales tales como una transformación afín y una convolución.

**X264**

Esta aplicación es un codificador de vídeo H.264 / AVC (Advanced Video Coding). H.264 describe la compresión con pérdida de un flujo de vídeo y también es parte de la norma ISO / IEC MPEG-4. La flexibilidad y la amplia gama de aplicación de la norma H.264 y su ubicuidad en los sistemas de video de próxima generación son las razones de la inclusión de x264 en el conjunto de pruebas PARSEC.

### 2.5.1 Blackscholes

Blackscholes es interesante por la importancia del problema que resuelve y la facilidad con la que puede ser paralelizada, por ello forma parte del repositorio de Intel RMS y del benchmark PARSEC. Esta aplicación es una de las más extendidas, pudiendo ser ejecutada en sistemas operativos Linux, Windows y Solaris 10. Soporta arquitecturas i386, Sparc y x86_64. La suite incluye las versiones paralelizadas utilizando los modelos Pthreads, OpenMP e Intel TBB.

#### *Información referente al modelo Black-Scholes*

Blackscholes [16] es utilizado en el ámbito de análisis financiero, esta aplicación resuelve el modelo Black-Scholes, modelo que gano el premio Nobel de Economía en 1997 y que se puede posicionar en una de las aportaciones más importante en este campo. Gracias a este modelo lo que se puede conseguir estimar el precio de una opción Europea en una fecha futura sin que esta tenga “riesgos”, todo esto bajo unos supuestos previamente tomados. Una opción financiera es un instrumento financiero derivado que se establece en un contrato. Este contrato da a su comprador el derecho pero no la obligación de comprar o vender bienes o valores a un precio predeterminado hasta una fecha concreta.

#### *Información necesaria y cálculo del modelo*

La aplicación Blackscholes calcula el precio estimado de un número de opciones financieras. Cada una de estas opciones se procesa de forma independiente y necesita unos parámetros para su cálculo [17]. Estos son:

- **P o C:** Tipo de operación, estas pueden ser compra (Call) o venta (Put).
- **S** : Precio del activo en el momento de la valoración
- **K**: Precio predeterminado para esa opción
- **R**: Tasa de interés en tiempo continuo
- **σ:** Volatilidad
- **T:** Tiempo restante hasta el vencimiento de la opción
- **F:** Función de distribución acumulativa de la distribución normal

Siendo la fórmula para la opción de compra:

$C(S,T)$= F (d1)S – F(d2)K$e^{-rT}$

En el que:

$$d1 = \frac{ln\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}$$

$$d2 = \frac{ln\left(\frac{S}{K}\right) + \left(r - \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}} = d1 - \sigma\sqrt{T}$$

Y siendo la fórmula para la opción de venta:

$P(S,T) = Ke^{-rT} - S + C(S,\ T) = F\ (-d2)\ Ke^{-rT} - F(-d1)S$

La aplicación coge cada uno de estos parámetros mencionados del fichero de entrada para cada una de las opciones, dependiendo del tipo de operación hace un cálculo u otro y vuelca el resultado al fichero de salida.

# 3. Planteamiento del problema

En este apartado se exponen una serie de requisitos referentes a la biblioteca de algoritmos a desarrollar. Con estos requisitos, se tiene como objetivo describir todas las características específicas que tiene que tener la biblioteca. Con el fin de estandarizar la biblioteca esta seguirá el estándar internacional C++ ISO/IEC 14882:2014.

## 3.1 Análisis de requisitos

Para conseguir que estos requisitos sean más claros serán expuestos en tablas con los siguientes campos:

- **Identificador**: Campo univoco que identifica solamente a un requisito.
- **Tipo**: Hace referencia al tipo de requisito que se trata, hay dos tipos:
  Funcional: Especifica lo que tiene que hacer el software
  No funcional: Específica cómo debe ser diseñado el software.
- **Prioridad**: Relevancia del requisito.
- **Conflicto**: Requisitos que no pueden llevarse a cabo, si este es implementado.
- **Descripción**: Breve explicación del contenido.
- **Justificación**: Objetivo con el que se le ha incluido.

A continuación están los requisitos que deben ser cumplidos.

| Identificador | REQF-1 | Tipo | Funcional |
|---|---|---|---|
| Descripción | La biblioteca de algoritmos debe contener 10 algoritmos pertenecientes a la STL de C++ y que pertenezcan a varias familias. | | |
| Justificación | Debido a la gran cantidad de algoritmos, se procederá a tratar un número considerable de ellos para demostrar la viabilidad de realizarlo con todos. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 13: REQF-1

| Identificador | REQF-2 | Tipo | Funcional |
|---|---|---|---|
| Descripción | La biblioteca de algoritmos debe contener una versión secuencial codificada de los algoritmos que no puede incluir la llamada al algoritmo de la STL. | | |
| Justificación | Con objetivo del entendimiento del algoritmo y su programación. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 14: REQF-2

| Identificador | REQF-3 | Tipo | Funcional |
|---|---|---|---|
| Descripción | La biblioteca de algoritmos debe contener una versión paralela de los algoritmos implementada con ISO Threads. | | |
| Justificación | Dar libertad al usuario en el modo de paralelización. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 15: REQF-3

| Identificador | REQF-4 | Tipo | Funcional |
|---|---|---|---|
| Descripción | La biblioteca de algoritmos debe contener una versión paralela de los algoritmos implementada con OpenMP. | | |
| Justificación | Dar libertad al usuario en el modo de paralelización. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 16: REQF-4

| Identificador | REQF-5 | Tipo | Funcional |
|---|---|---|---|
| Descripción | La biblioteca de algoritmos debe contener una versión paralela de los algoritmos implementada con TBB. | | |
| Justificación | Dar libertad al usuario en el modo de paralelización. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 17: REQF-5

| Identificador | REQF-6 | Tipo | Funcional |
|---|---|---|---|
| Descripción | Los argumentos pasados a los algoritmos son los mismos que la versión de la STL, añadiéndole la versión del algoritmo y el número de hilos en los que se va a paralelizar. | | |
| Justificación | Respetar la interfaz original de los algoritmos y utilizar solo una interfaz para todas las versiones. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 18: REQF-6

| Identificador | REQF-7 | Tipo | Funcional |
|---|---|---|---|
| Descripción | Las versiones de cada algoritmo serán implementadas con el objetivo de que las no utilizadas sean eliminadas al compilar. | | |
| Justificación | Evitar sobrecarga y proporcionar sobrecarga de funciones separadas y especializaciones dependiendo del modo de paralelización. | | |
| Prioridad | Media | Conflicto | # |

Tabla 19: REQF-7

| Identificador | REQNF-1 | Tipo | No funcional |
|---|---|---|---|
| Descripción | La implementación, pruebas y análisis de resultados serán realizadas en el SO Linux. | | |
| Justificación | Unificar y estandarizar el entorno de desarrollo y evaluación. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 20: REQNF-1

| Identificador | REQNF-2 | Tipo | No funcional |
|---|---|---|---|
| Descripción | El análisis de resultados de todo algoritmo se realiza por medio de microbenchmarks. | | |
| Justificación | Debido al gran número de algoritmo es inviable realizar un análisis de resultados en aplicaciones reales. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 21: REQNF-2

| Identificador | REQNF-3 | Tipo | No funcional |
|---|---|---|---|
| Descripción | Evaluar cada una de las aplicaciones de PARSEC y se elegirá una en la que se pueda utilizar algún de los algoritmos de la biblioteca. | | |
| Justificación | Conseguir una aplicación de la que se pueda hacer un uso sencillo para poner a prueba nuestros algoritmos. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 22: REQNF-3

| Identificador | REQNF-4 | Tipo | No funcional |
|---|---|---|---|
| Descripción | Se realizará un análisis de resultados de un algoritmo sobre un caso práctico, que comprenderá una aplicación del benchmark PARSEC. | | |
| Justificación | Poner a prueba la implementación en un entorno real como los proporcionados por el benchmark PARSEC. | | |
| Prioridad | Media | Conflicto | # |

Tabla 23: REQNF-4

| Identificador | REQNF-5 | Tipo | No funcional |
|---|---|---|---|
| Descripción | En el caso de la que la aplicación del benchmark estuviese escrita en lenguaje C, se procedería a modernizar el código a C++. | | |
| Justificación | Varias aplicaciones de PARSEC están escritas en lenguaje C por lo tanto no se podría utilizar nuestra biblioteca. | | |
| Prioridad | Media | Conflicto | # |

Tabla 24: REQNF-5

| Identificador | REQNF-6 | Tipo | No funcional |
|---|---|---|---|
| Descripción | Toda compilación deber ser realizada con el compilador GCC. | | |
| Justificación | Unificar y estandarizar el entorno de desarrollo y evaluación. | | |
| Prioridad | Alta | Conflicto | # |

Tabla 25: REQNF-6

# 4. Diseño e implementación

La primera parte de la biblioteca está compuesta por la declaración de cuatro estructuras vacías, una para cada una de las políticas que puede tener un algoritmo: secuencial, ISO Threads, OpenMP y TBB. Luego se realiza una instanciación de cada una de ella para que sean usadas por el usuario al hacer la llamada al algoritmo. Con estas estructuras y la función enable_if es posible conseguir una misma declaración de la función para cada una de las políticas. La sentencia enable_if comprueba de que tipo es el primer argumento del algoritmo llamado, que corresponde a la política, dependiendo que política sea se hará referencia a un código o a otro, todo esto ocurriendo en tiempo de compilación. Gracias a esto la estructura del código es más clara y se consigue una interfaz unificada que para los usuarios es beneficioso.

Todos los algoritmos reciben por argumento los mismos parámetros que su versión original de la STL, incluyendo la política que será el primer argumento del algoritmo y el número de hilos que será el último.

Todas las versiones tienen un control del número de hilos ya que nunca puede ser negativo o por ejemplo en la política secuencial tiene que ser 1.

Para que la explicación sea más clara y como todo algoritmo recibe un iterador de comienzo y otro de final, hablaremos de contenedor al hacer referencia al conjunto de elementos que engloba los iteradores intermedios del iterador de comienzo con el de final. Cuando se nos pase un segundo iterador de comienzo lo tomaremos como otro contenedor que termina cuando el iterador se itera tantas veces como el primero.

## 4.1 Implementación algoritmo count

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final de contenedor y el valor a contar.

### 4.1.1 Versión secuencial

Esta versión secuencial está compuesta por un bucle for, que recorre todo el contenedor por medio de un iterador que hace de índice, y comprueba si el iterador contiene el valor que se quiere buscar. Si coinciden se incrementa una variable previamente declarada e inicializada a cero, que será la devuelta al recorrer todo el contenedor.

### 4.1.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos y una variable atómica donde se almacenara el resultado. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el repartimiento de hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor que le toca analizar a cada hilo. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función lamda en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor y dicho resultado se suma a la variable atómica. Para saber el segmento se calcula el iterador de comienzo y final para una porción del contenedor.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno y cuando estos acaban devuelve la variable atómica que tendrá la suma de todos los hilos.

### 4.1.3 Versión con OpenMP

Esta versión es muy parecida a la versión secuencial, de hecho el código es exactamente el mismo con una directiva de OpenMP incluida antes del bucle.

Esta línea es un parallel for reduction a la que se le indica la variable donde se va a almacenar los resultados y el número de hilos. Esta directiva hace un repartimiento de las interacciones del bucle en hilos y el resultado de cada uno se termina sumando a la variable donde se ha indicado que se almacene el resultado. De esta forma se realizaría algo muy similar a los realizado con ISO Threads pero de una forma muy abstracta y desentendiéndonos de gestionar el acceso a la variable común.

### 4.1.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init. Después procedemos hacer un return de la llamada parallel_reduce que recibe por argumento:

- Un blocked_range inicializado con el comienzo del contenedor y el final, al realizar begin del blocked_range recibiremos el comienzo del contenedor y al hacer end el final.
- El valor con el que comienza el reduce en este caso 0.
- Una función lamda en la que recibe por argumento el blocked_range y la variable donde se almacena el resultado. En dicha función se realiza un for por medio de un iterador que es inicializado al comienzo del blocked_range y que se incrementa hasta el final del blocked_range. En dicho bucle realizamos la comprobación de si el iterador contiene el valor deseado se incrementa la variable de resultado. Después se hace return del resultado.
- Una función landa que recibe dos parámetros del mismo tipo que donde se almacena el resultado y devuelve la suma de ellos.

Con todo esto se realiza algo parecido a lo realizado con OpenMP y se crean hilos para cada parte del bucle que itera el blocked_range y luego se realiza la suma de los resultados.

## 4.2 Implementación algoritmo find

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor y el valor a buscar.

### 4.2.1 Versión secuencial

Esta versión secuencial está compuesta por un bucle for, que recorre todo el contenedor por medio de un iterador que hace de índice y comprueba si el iterador contiene el valor que se quiere buscar, que es pasado por argumento. Si coinciden se hace return de ese iterador.

### 4.2.2 Versión con ISO Threads

En esta versión lo primero que se hace es calcular la distancia entre los iteradores de comienzo y final que han sido pasados por argumento. Seguidamente nos creamos dos vectores uno de futuros y otro de iteradores que serán los iteradores de fin de cada uno de los segmentos.

A continuación un bucle for que iterará el mismo número de veces que numero de hilos se quiera en la ejecución. En dicho bucle calculamos el comienzo y final de cada segmento y añadimos al vector de iteradores el final del él. Seguidamente nos creamos un packaged_task que realizara la versión secuencial para el segmento previamente calculado. De dicho packaged_task cogemos su futuro y lo añadimos al vector de futuros. Por último lanzamos un hilo que realice el packaged_task y hacemos detach de dicho hilos.

Con esto disponemos en el vector de futuros los resultados obtenidos por cada hilo y también si no han encontrado resultado el que devolverá que será el final del segmento. Gracias a esto para localizar el primer elemento que se ha encontrado si hay alguno, con un bucle pasamos por todos los futuros viendo el resultado que ha obtenido haciendo wait de dicho futuro previamente para asegurarnos que está el resultado. Si el resultado es igual al final del segmento de dicho hilo procedemos al siguiente futuro pero si es distinto devolvemos dicho iterador que corresponderá con el primer iterador que contiene el elemento buscado. Si no existía dicho valor en el contenedor se devuelve el iterador final del contenedor.

### 4.2.3 Versión con OpenMP

En esta versión necesitamos un vector de iteradores, que serán los resultados de cada hilo. Este vector tiene tantas posiciones como hilos se quieran ejecutar y dicha posición esta inicializada al iterador final del contenedor que es pasado por argumento. A continuación ponemos la directiva de sección paralela indicándole cuantos hilos se van a ejecutar. Dentro de esta obtenemos el identificador del hilo por medio de omp_get_thread_num que nos servirá para añadir el resultado de cada hilo

en el vector de resultados y poder saber el orden que tiene cada resultado en el contenedor, dado que la posición 0 corresponderá al primer hilo lanzado. Después llamamos a la directiva omp parallel for y seguido tenemos el mismo bucle que en la versión secuencial a diferencia que dentro de él comprobamos si el iterador contiene el valor que buscamos y si no se ha añadido ya un iterador al vector de resultados para ese hilo, esto se realiza comparando que la posición de resultados de ese hilo tenga valor distinto que el ultimo iterador del contenedor. Si el contenido es igual y no se ha añadido ninguno antes, en la posición de resultados de ese hilo se pone el iterador encontrado.

Fuera de la sección paralela nos recorremos el vector de resultados y comprobamos si alguna posición tiene un iterador distinto al último del contenedor, si es así devolvemos dicho iterador y si ninguno contiene uno diferente devolvemos el iterador final del contenedor. Como nos recorremos el vector ordenadamente sabemos el orden que tienen en el contenedor los resultados pudiendo devolver el primero que este en él.

### 4.2.4 Versión con TBB

En esta versión lo primero que se realiza es declararnos un vector de iteradores donde almacenaremos los resultados y un mutex, después indicaremos el número de hilos a TBB por medio del task_scheduler_init. Después procedemos hacer una llamada a parallel_for que recibe por argumento:

- Un blocked_range inicializado con el comienzo del contenedor y el final
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for por medio de un iterador que es inicializado al comienzo del blocked_range y que se incrementa hasta el final del blocked_range. Dentro del bucle se compara si el contenido del iterador que recorre el contenedor tiene el contenido que es pasado por argumento, si es así, mete en el vector de resultados dicho iterador. El acceso al vector está protegido por el mutex para evitar que dos hilos accedan al vector de resultados al mismo tiempo.

Una vez que se ha analizado todo el contenedor, nos recorremos el vector de resultados buscando cual es el iterador resultado más próximo al comienzo del contenedor, esto lo averiguamos por medio de la función distance.

Por ultimo devolvemos el iterador más próximo al comienzo y en el caso de ningún resultado devolveremos el último del contenedor.

## 4.3 Implementación algoritmo copy

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor origen y el iterador de comienzo del contenedor de destino.

### 4.3.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo el contenedor de origen y al mismo tiempo el contenedor de destino por medio de iteradores que son los que recibe por argumento. Dentro de este bucle se va igualando el contenido del iterador de destino al de origen.

### 4.3.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor de origen tanto el inicio como el final y además el comienzo del segmento del contenedor de destino. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor de origen y la posición inicial del segmento del contenedor de destino.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno.

### 4.3.3 Versión con OpenMP

Esta versión es muy parecida a la secuencial pero se han realizado varios cambios. En esta lo primero que se calcula es la distancia entre los iteradores de comienzo y final del contenedor de origen. A continuación hay un bucle for que se ejecutará un número de veces igual que la distancia que se ha calculado previamente. Para pasar por todas las posiciones en cada iteración se iguala el contenido del iterador de comienzo de destino más el número de iteración al contenido del iterador de comienzo del contenedor de origen más el número de iteración. Luego al bucle le añadimos la directiva parallel for indicando el número de hilos.

Esta forma de recorrernos los contenedores ha sido necesaria ya que las directivas OpenMP de parallel for solo pueden tener en cuenta un índice del bucle y en este caso tendríamos que ir iterando respecto dos iteradores.

### 4.3.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init y calcular la distancia entre iteradores del contenedor de

origen como en la versión de OpenMP. Después procedemos la llamada parallel_reduce que recibe por argumento:

- Un blocked_range inicializado con 0 y el valor de la distancia calculada.
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for en el que se itera a partir de un int, que se inicializa con el contenido del comienzo del bloked y se ejecuta hasta el contenido del final de bloked. De esta forma tenemos un bucle for que itera un número de veces igual a la distancia. Dentro de este bucle en cada iteración se iguala el contenido del iterador de destino más el número de iteración al contenido del iterador de comienzo del contenedor de origen más el número de iteración.

## 4.4 Implementación algoritmo transform

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor de origen, el iterador de comienzo del contenedor de destino y la función que se aplica al origen y se almacena en el contenedor de destino.

### 4.4.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo el contenedor de origen y el contenedor de destino por medio de iteradores. Dentro de este bucle se iguala el contenido del iterador de destino a la función pasada por argumento con el contenido del iterador de origen pasado a dicha función.

### 4.4.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor de origen tanto el inicio como el final y además el comienzo del segmento del contenedor de destino. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor de origen y la posición inicial del segmento del contenedor de destino y la función que es pasada por argumento.
Una vez lanzados todos los hilos el hilo maestro espera por cada uno de ellos.

### 4.4.3 Versión con OpenMP

Esta versión es muy parecida a la secuencial pero se han realizado varios cambios. En esta lo primero que se calcula es la distancia entre los iteradores de comienzo y final

del contenedor de destino. A continuación hay un bucle for que se ejecutará un número de veces igual que la distancia previamente calculada. Para pasar por todas las posiciones en cada iteración se iguala el contenido del iterador de comienzo de destino más el número de iteración a la función que se ha pasado por argumento. Y a esta función se le pasa por argumento el contenido del iterador de comienzo del contenedor de origen más el número de iteración.

Luego al bucle le añadimos la directiva parallel for indicando el número de hilos.

### 4.4.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init y calcular la distancia entre iteradores del primer contenedor como en la versión de OpenMP. Después procedemos la llamada parallel_for que recibe por argumento:

- Un blocked_range inicializado con 0 y el valor de la distancia calculada.
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for en el que se itera a partir de un int, que se inicializa con el contenido del comienzo del bloked y se ejecutara hasta el contenido del final de bloked. De esta forma tenemos un bucle for que itera un número de veces igual a la distancia. Dentro de este bucle en cada iteración se iguala el contenido del iterador de destino más el número de iteración a la función que se ha pasado por argumento. Y a esta función se le pasa por argumento el contenido del iterador de comienzo del contenedor de origen más el número de iteración.

## 4.5 Implementación algoritmo move

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor de origen y el iterador de comienzo del contenedor de destino.

### 4.5.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo el contenedor de origen y al mismo tiempo el contenedor de destino por medio de iteradores que son lo que recibe por argumento. Dentro de este bucle se hace un move del contenido del iterador de origen y se iguala al de destino.

### 4.5.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor de origen tanto el inicio como el final y además el comienzo del segmento del contenedor de destino. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor de origen y la posición inicial del segmento del contenedor de destino.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno de ellos.

### 4.5.3 Versión con OpenMP

Esta versión es muy parecida a la secuencial pero se han realizado varios cambios. En esta lo primero que se calcula es la distancia entre los iteradores de comienzo y final del contenedor de origen. A continuación hay un bucle for que se ejecutara un número de veces igual que la distancia previamente calculada. Para pasar por todas las posiciones en cada iteración se iguala el contenido del iterador de comienzo del destino más el número de iteración al resultado de hacer move del iterador de comienzo del contenedor de origen más el número de iteración.

Luego al bucle le añadimos la directiva parallel for indicando el número de hilos.

### 4.5.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init y calcular la distancia entre iteradores del contenedor de origen como en la versión de OpenMP. Después procedemos la llamada parallel_reduce que recibe por argumento:

- Un blocked_range inicializado con 0 y el valor de la distancia calculada.
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for en el que se itera a partir de un int, que se inicializa con el contenido del comienzo del bloked y se ejecuta hasta el contenido del final de bloked. De esta forma tenemos un bucle for que itera un número de veces igual a la distancia. Dentro de este bucle en cada iteración se iguala el contenido del iterador de destino más el número de iteración al resultado de hacer move del iterador de comienzo del contenedor de origen más el número de iteración.

## 4.6 Implementación algoritmo fill

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor y el valor que van a tomar todas las posiciones.

### 4.6.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo el contenedor por medio de iteradores que son lo que recibe por argumento. Dentro de este bucle se iguala el contenido del iterador al valor pasado por argumento.

### 4.6.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno y cuando estos acaban devuelve la variable atómica que tendrá la suma de todos los hilos.

### 4.6.3 Versión con OpenMP

En esta versión a diferencia de la secuencial hemos cambiado el bucle while por un bucle for, para poder hacer uso de la directiva parallel for. Por lo tanto quedaría un bucle for que se recorre el contenedor. Y en dicho bucle se iguala el contenido del iterador que hace de índice al valor pasado por argumento.

A este for se le añade arriba la directiva parallel for indicando el número de hilos y con esto se realizaría la paralización del algoritmo.

### 4.6.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init. Después procedemos hacer un return de la llamada parallel_for que recibe por argumento:

- Un blocked_range inicializado con el comienzo del contenedor y el final
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for por medio de un iterador que es inicializado al comienzo del blocked_range y que se incrementa hasta el final del blocked_range. Dentro del bucle se iguala el contenido del iterador que hace de índice al valor pasado por argumento.

## 4.7 Implementación algoritmo generate

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del contenedor y la función que se va ejecutar para cada posición del contenedor y cuyo resultado será almacenado en las posiciones del contenedor.

### 4.7.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo el contenedor por medio de iteradores que son lo que recibe por argumento. Dentro de este bucle se iguala el contenido del iterador a la función pasada por argumento.

### 4.7.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno de ellos.

### 4.7.3 Versión con OpenMP

En esta versión a diferencia de la secuencial hemos cambiado el bucle while por un bucle for, para poder hacer uso de la directiva parallel for. Por lo tanto quedaría un bucle for que se recorre el contenedor. Y en dicho bucle se iguala el contenido del iterador a la función pasada por argumento.
A este for se le añade arriba la directiva parallel for indicando el número de hilos y con esto se realizaría la paralización del algoritmo.

### 4.7.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init. Después procedemos hacer un return de la llamada parallel_for que recibe por argumento:

- Un blocked_range inicializado con el comienzo del contenedor y el final
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for para iterar en el blocked_range que contiene los iteradores del contenedor, y en dicho bucle realizamos la igualación del contenido del iterador a la función que nos han indicado por argumento.

## 4.8 Implementación algoritmo swap_ranges

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del primer contenedor y el iterador de comienzo del segundo contenedor.

### 4.8.1 Versión secuencial

Esta versión secuencial de este algoritmo está compuesta por un bucle while que se va recorriendo los dos contenedores por medio de iteradores que son lo que recibe por argumento. Dentro de este bucle se hace un iter_swap de los dos iteradores que están recorriendo los contenedores.

### 4.8.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el reparto del contenedor a los hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor de origen tanto el inicio como el final y además el comienzo del segmento del segundo contenedor. Una vez sabiendo los segmentos a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del primer contenedor y del segundo contenedor.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno y cuando estos acaban devuelve la variable atómica que tendrá la suma de todos los hilos.

### 4.8.3 Versión con OpenMP

Esta versión es muy parecida a la secuencial pero se han realizado varios cambios. En esta lo primero que se calcula es la distancia entre los iteradores de comienzo y final del primer contenedor. A continuación hay un bucle for que se ejecuta un número de veces igual que la distancia previamente calculada. Para pasar por todas las posiciones en cada iteración se realiza la operación iter_swap del iterador de comienzo del primer contenedor más el número de iteración y del iterador de comienzo del segundo contenedor más el número de iteración.

Luego al bucle le añadimos la directiva parallel for indicando el número de hilos. Esta forma de recorrernos los contenedores ha sido necesaria ya que las directivas OpenMP de parallel for solo pueden tener en cuenta un índice del bucle y en este caso tendríamos que ir iterando respecto dos iteradores.

### 4.8.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init y calcular la distancia entre iteradores del primer contenedor como en la versión de OpenMP. Después procedemos la llamada parallel_for que recibe por argumento:

- Un blocked_range inicializado con 0 y el valor de la distancia calculada.
- Una función lamda en la que recibe por argumento el blocked_range. En dicha función se realiza un for en el que se iterara a partir de un int, que se inicializa con el contenido del comienzo del bloked y se ejecuta hasta el contenido del final de bloked. De esta forma tenemos un bucle for que itera un número de veces igual a la distancia. Dentro de este bucle en cada iteración se realiza la operación iter_swap del iterador de comienzo del primer contenedor más el número de iteración y del iterador de comienzo del segundo contenedor más el número de iteración.

## 4.9 Implementación algoritmo accumulate

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del primer contenedor y el valor en el que comienza la suma.

### 4.9.1 Versión secuencial

Esta versión secuencial está compuesta por un bucle for, que recorre todo el contenedor por medio de un iterador que hace de índice, dentro de este en una variable pasada por argumento se van acumulando los valores de cada posición del contenedor y luego se devuelve el resultado acumulado.

### 4.9.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos y un mutex para proteger el acceso cuando cada hilo acceda a la variable donde se almacena el resultado. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el repartimiento de hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del contenedor que le toca analizar a cada hilo. Una vez sabiendo el segmento a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese

segmento del contenedor, almacenando el resultado en una variable local, luego esa variable local es sumada a la variable que es pasada por argumento que contiene el comienzo de la acumulación, esta suma está protegida por el mutex para que ningún hilo modifique a la vez dicha variable.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno y cuando estos acaban devuelve la variable donde se ha realizado la acumulación.

### 4.9.3 Versión con OpenMP

En esta versión lo primero que se hace es declararnos un vector de resultados de tamaño igual al número de hilos. Después un bucle for con la sentencia parallel for que se recorre todo el contenedor y cada hilo ira acumulando los valores que vaya leyendo del contenedor en una posición del vector de resultados. La posición de cada uno se sabe con la sentencia de omp_get_num_thread que nos da el identificador del hilo en ejecución.

Seguidamente ya en secuencial hay un bucle que se recorre el vector de resultados y va acumulando los valores de este en la variable que tenía el comienzo de la acumulación. Por último se devuelve esta variable.

Para esta versión hubiese sido más sencillo utilizar la sentencia parallel for reduction como en el count, pero OpenMP no lo soporta para tipos no básicos, como puede ser una clase creada por el usuario.

### 4.9.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init. Después procedemos a hacer un return de la llamada parallel_reduce que recibe por argumento:

- Un blocked_range inicializado con el comienzo del contenedor y el final
- El valor con el que comienza el reduce en este caso la variable que es pasada por argumento.
- Una función lamda en la que recibe por argumento el blocked_range y la variable que donde se almacena el resultado. En dicha función se realiza un for para iterar en el blocked_range que contiene los iteradores del contenedor, y en dicho bucle se suma el valor del contenido del iterador a la variable donde se almacena el resultado. Despues se hace return del resultado.
- Una función landa que recibe dos parámetros del mismo tipo que donde se almacena el resultado y devuelve la suma de ellos.

## 4.10 Implementación algoritmo inner_product

Este algoritmo originalmente recibe dos iteradores uno de comienzo y otro de final del primer contenedor, un iterador de comienzo del segundo contenedor y el valor en el que comienza el resultado.

### 4.10.1 Versión secuencial

Esta versión secuencial está compuesta por un bucle while, que se recorre los dos contenedores simultáneamente por medio de dos iteradores. En dicho bucle se va acumulando en una variable pasada por argumento la multiplicación del contenido de los dos iteradores.

### 4.10.2 Versión con ISO Threads

En esta versión lo primero que se realiza es declarase un vector de hilos y un mutex para proteger el acceso cuando cada hilo acceda a la variable donde se almacena el resultado. Después se procede a saber el número de elementos que contiene el contenedor necesario para hacer el repartimiento de hilos.

A continuación se crea un bucle que iterará un número de veces igual al número de hilos. En este bucle se calcula el segmento del primer contenedor que le toca analizar a cada hilo y la primera posición de los segmentos del segundo contenedor. Una vez sabiendo los segmentos a analizar se lanza un hilo por medio de una función landa en el que se llama a la versión secuencial del algoritmo para ese segmento del contenedor almacenando dicho valor en una variable local, luego esa variable local es sumada a la variable que es pasada por argumento que contiene el valor con el que comienza el resultado, esta suma está protegida por el mutex para que ningún hilo modifique a la vez dicha variable.

Una vez lanzados todos los hilos el hilo maestro espera por cada uno y cuando estos acaban devuelve la variable atómica que tendrá la suma de todos los hilos.

### 4.10.3 Versión con OpenMP

En esta versión lo primero que se calcula es la distancia entre los iteradores de comienzo y final del primer contenedor y se declara un vector de resultados con tamaño igual al número de hilos. A continuación hay un bucle for con la directiva parallel for que se ejecuta un número de veces igual que la distancia previamente calculada. Para pasar por todas las posiciones en cada iteración se acumula en la posición del vector de resultados correspondiente a cada hilo el resultado la multiplicación del contenido del iterador resultante de sumarle al iterador de comienzo del primer contenedor el número de iteración y el contenido del iterador resultante de sumarle al iterador de comienzo del segundo contenedor el número de

iteración. Para saber la posición de cada hilo usamos la sentencia de omp_get_num_thread que nos devuelve el identificador de cada hilo

A continuación se recorre el vector de resultados acumulando los valores que contiene en la variable de comienzo de acumulación pasada por argumento y devolvemos dicha variable.

No ha sido posible utilizar parallel reduction debido que OpenMP no soporta reduction con variables de tipo no básico.

### 4.10.4 Versión con TBB

En esta versión lo primero que se realiza es indicar el número de hilos a TBB por medio del task_scheduler_init y calcular la distancia entre iteradores del primer contenedor como en la versión de OpenMP. Después procedemos a hacer un return de la llamada parallel_reduce que recibe por argumento:

- Un blocked_range inicializado con 0 y el valor de la distancia calculada.
- El valor con el que comienza el reduce en este caso la variable que es pasada por argumento.
- Una función lamda en la que recibe por argumento el blocked_range y la variable que donde se almacena el resultado. En dicha función se realiza un for en el que se iterara a partir de un int, que se inicializa con el contenido del comienzo del bloked y se ejecutara hasta el contenido del final de bloked. De esta forma tenemos un bucle for que itera un número de veces igual a la distancia. Dentro de este bucle en cada iteración acumulamos en la variable donde se almacena el resultado la multiplicación del contenido del iterador resultante de sumarle al iterador de comienzo del primer contenedor el número de iteraciones y el contenido del iterador resultante de sumarle al iterador de comienzo del segundo el número de iteraciones.
- Una función landa que recibe dos parámetros del mismo tipo que donde se almacena el resultado y devuelve la suma de ellos.

Una vez implementados todos los algoritmos hay que hacer algunas aclaraciones sobre ellos, y dependiendo de la versión de la que se trate. En el caso del *find* para la versión de ISO Threads será más rápida que las otras dos versiones, esto es debido a que si el primer hilo encuentra resultados automáticamente se devuelve el resultado olvidando los demás hilos, esto en OpenMP y TBB no es posible ya que estos modelos de paralelización no nos dan tanta gestión sobre los hilos como ISO Threads. Por lo tanto las versiones de TBB y OpenMP siempre tardaran el mismo tiempo independiente de la posición del objeto buscado en el contenedor. Por otro lado TBB no nos da mecanismo

de exclusión mutua o de protección respecto a una variable por lo tanto en el caso de find es necesario hacer uso de mutex de ISO Threads para evitar que varios hilos accedan al vector de resultados al mismo tiempo.

# 5. Caso práctico: Blackscholes

En este aparto se va explicar la estructura general de la aplicación blackscholes original y los cambios realizados para una modernización del código a C++, donde poder utilizar alguno de nuestros algoritmos de la biblioteca.

La aplicación de blackscholes lo que realiza a grandes rasgos es leer de un fichero donde cada línea representa una opción financiera a someter a la fórmula de blackscholes, compuesta por todos los elementos necesarios. Luego somete a la formula cada una de las opciones financieras que ha leído y vuelca los resultados a otro fichero. El fichero de entrada y de salida es pasado por argumento al ejecutar la aplicación. El fichero de entrada debe contener en la primera línea el número de opciones que contiene.

## 5.1 Blackscholes original

La aplicación original lo primero que realiza es crearse globalmente un tipo de estructura llamado OptionData que contiene las variables necesarias para la formula, en este caso serían floats para todas menos el indicador de si es venta o compra que es un char.

Como variables globales se crean un puntero de tipo OptionData y cinco punteros a float que cada uno representa a cada una de las variables necesarias, un puntero a int para el tipo de opción y un puntero a float para guardar los resultados de cada opción.

En la función main lo primero que se hace es crearse un puntero a un fichero, abre el fichero que contiene las opciones, obtiene el número de opciones que tiene el fichero y hace una reserva de ese número de opciones para el puntero de OptionData y de resultados. Seguidamente con un bucle se recorre todo el fichero leyendo con fscanf y metiendo en el array de OptionData cada una de las opciones leídas. Seguidamente los parámetros leídos de cada una de las opciones que están en las OptionData, son pasados a los arrays para cada uno de los parámetros (punteros a float), forzando por medio de buffers auxiliares que los parámetros del mismo tipo de todas las opciones estén alineados en memoria.

A continuación se llama a una función llamada bs_thread en el que va llamando por medio de un bucle for a la función BlkSchlsEqEuroNoDiv para cada una de las opciones, pasándole por argumento los parámetros necesarios que son cogidos de los arrays alineados. A continuación guarda el resultado en el array de resultados para cada opción. La función BlkSchlsEqEuroNoDiv realiza la fórmula de blackscholes apoyándose en la función CNDF que forma parte de la formula. Bs_thread además tiene un bucle exterior que se ejecuta por defecto 100 veces que engloba el bucle anteriormente citado, consiguiendo que el coste de cómputo sea más elevado.

Una vez teniendo todos los resultados en el array se vuelcan los resultados al fichero de salida por medio de fprintf diciendo que se imprima con 18 decimales.

## 5.2 Blackscholes modernizado.

En un primer momento para cambiar la versión original y pasarla a C++ se cambiaron los define a contexpr y los fptype  a templates, además todos los arrays que había han sido convertidos en vectores de la STL. Además el manejo de ficheros en vez de realizarse con llamadas como fopen, fscanf o fprintf, se hace con ifstream y oftream, con esto el manejo es más claro y sencillo.

Una vez realizado estos cambios analizando la aplicación nos damos cuenta que el algoritmo de la STL que podemos utilizar es *transform*. Ya que hay que aplicar una función a todas las posiciones de un vector y almacenar los resultados en otro vector. Por lo tanto hemos tenido que cambiar la función BlkSchlsEqEuroNoDiv para que reciba por argumento una estructura o clase que contenga todos los argumentos que tenía previamente.

Para que se pueda ejecutar el algoritmo *transform* perfectamente hemos generado dos versiones de blackscholes uno en el que se almacenan los parámetros de cada opción financiera en una estructura de vectores y otra que es un vector de estructuras.

### 5.2.1 Estructura de vectores

En esta versión además de la estructura que estaba en la original nos hemos creado otra llamada OptionDataInVectors en el tenemos un vector para cada uno de los parámetros de la formula. Para que *transform* pueda trabajar con esta estructura hemos tenido que implementar un iterador para esa estructura, para poder iterar en los vectores que contiene simultáneamente. OptionDataInVectors tiene las funciones de begin y end como todo contenedor que devolverán un iterador que hará referencia a los primeros elementos.

Para el iterador nos hemos creado la estructura OptionDataInVectors_Iter que hace de iterador. Esta estructura tiene que tener los operadores necesarios para su uso semejantes a los de un iterador de acceso aleatorio. Este iterador tiene como contenido un iterador de tipo vector de todos los vectores que contenía la estructura OptionDataInVectors. Los operadores implementados para el iterador son los de acceso, los de incremento y decremento, los de incremento y decremento pero de un número, no de uno en uno, el de suma y resta de un número a él mismo, el de asignación, el de igualdad y desigualdad y el de mayor, menor, mayor o igual, menor o igual que otro iterador. Todos estos operadores han sido programados haciendo ese mismo operador sobre los iteradores de los vectores o comparando estos. Para el caso del operador de acceso lo que se realiza es devolver una estructura llamado OptionDataValor, que contiene cada uno de los parámetros necesarios, y que al

acceder a iterador esta estructura estará rellena con los valores de la opción financiera que se estuviese.

Con todo esto lo cambios a realizar respecto al original, solo hay que declararse el vector de OptionData para seguir obteniendo los datos del fichero y el vector de resultados. En el main declarar un OptionDataInVectors y en este rellenar los vectores que contiene a partir del vector OptionData. Después llamar al bs_thread pasándole por argumento el OptionDataInVectors declarado. Otro cambio realizado es ese, que bs_thread recibe un OptionDataInVectors.

Seguidamente en el bs_thread se elimina el bucle interno (no el de aumento de computo) que había para pasar por todas las posiciones y solo incluimos la llamada a la función *transform* de la biblioteca creada, pasándole el argumento de política que queramos, el comienzo y el final del OptionDataInVectors por medio de las funciones begin y end, el comienzo del vector donde se almacenan los resultados, la función BlkSchlsEqEuroNoDiv modificada que recibirá por argumento un OptionDataValor y el número de hilos con el que se ejecutará.

### 5.2.2 Vector de estructuras

En esta versión nos hemos tenido que crear una clase que hemos llamado Data que contiene los atributos necesarios para la formula. En esta clase nos hemos creado el constructor donde se le pasa por argumento los valores que van a tomar cada una de los atributos.

En el main se declara un vector de Data y sigue siendo igual la parte que se almacena los valores en el vector de OptionData. Después con los valores que se tienen en el vector de OptionData se rellena el vector de Data por medio de push_back, pasándole por argumento el objeto a almacenar inicializado en el constructor los atributos de la opción financiera que sea.

En este caso BlkSchlsEqEuroNoDiv recibe por argumento un objeto Data y bs_thread recibe por argumento un vector de Data. Dentro del bs_thread se suprime el bucle interno que había y solo incluimos la llamada a la función *transform* de la biblioteca creada, pasándole el argumento de política que queramos, el comienzo y el final del el vector de Data por medio de las funciones begin y end, el comienzo del vector donde se almacenan los resultados, la función BlkSchlsEqEuroNoDiv modificada y el número de hilos con el que se ejecutará.

# 6. Evaluación de resultados

En este apartado se evalúa el rendimiento que ofrecen los algoritmos que se han programado, para esto se va a realizar de dos modos, primeramente creando microbenchmarks que son programas sencillos con los que comprobar el rendimiento y luego en un caso práctico se evaluará el rendimiento de un algoritmo en concreto. Los benchmarks no dan fiabilidad 100% de los resultados, es decir que un benchmarks nos diga que cierta versión de un algoritmo es la mejor no tiene por qué ser verdad para todos los casos. Ya que dependiendo del problema puede ser que una versión que es la mejor en otra situación pase a ser la peor. De todas formas en estos casos se quiere demostrar la mejora que existe, no determinar explícitamente que versión es mejor para cada algoritmo. También es cierto que debido a limitaciones de implementación hay ciertas versiones que tienen limitaciones funcionales, estas serán comentadas al evaluar cada algoritmo.

Las pruebas han sido realizadas en un computador con procesador Intel i7-472HQ con 4 núcleos físicos a 2.3Ghz e hyper-threading lo que supone 8 núcleos lógicos, además este ordenador tiene 8GB de memoria RAM. Las tomas de tiempos han sido realizadas por medio del kernel de linux perf [18] que además de tomar tiempos nos permite ver contadores hardware de la ejecución de un comando.

## 6.1 Evaluación de microbenchmarks

Con objetivo de evaluar el rendimiento de los algoritmos se han implementado pequeños programas sencillos en los que hacer uso de los algoritmos, como las operaciones con tipos básicos son muy rápidas y fácilmente optimizadas por el compilador, las pruebas han sido hechas respecto a una clase creada, que contiene un int y un float y que tiene sobrecargados los operadores:

- Suma: Donde al sumar dos objetos de la misma clase se suman los int por un lado y por otro los floats.
- Igualdad: Se comparan los dos atributos de las clases y si son ambos iguales es que son iguales.
- Asignación: Donde se le asigna valor a un nuevo objeto a partir de otro dándole valor a los atributos.

Todos estos operadores salvo la suma contienen una sobrecarga de cómputo para simular que dichas operaciones tardan más en ejecutarse. Además como es necesario tiene los constructores de la clase para poder inicializar los objetos.

### 6.1.1 Count

El gráfico (ilustración 6) muestra como todas las versiones dan buenos resultados. Y en todos los casos salvo cuando es con 4 hilos todas las versiones dan speedups muy parejos. En el caso de 4 hilos destaca TBB ligeramente por encima de OpenMP y TBB ligeramente sobre ISO Threads.

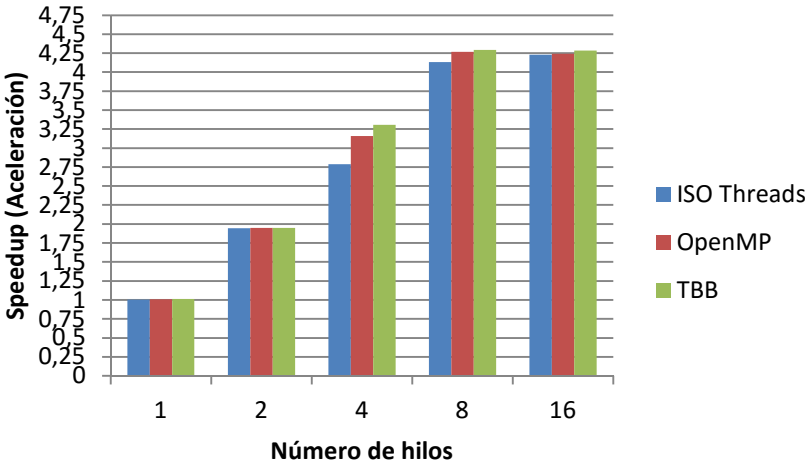



Ilustración 6: Gráfico de aceleraciones de Count

### 6.1.2 Find

En el gráfico (ilustración 7) se puede ver de nuevo como todas las versiones ofrecen buenos resultados, y como vuelve a ocurrir lo mismo que en el anterior algoritmo, los speedups son muy parejos en todos los casos menos con 4 hilos que ISO Threads queda más descolgada respecto a las demás.

En esta prueba se han tomado medidas del peor caso, en el que no hay coincidencias en la búsqueda y hay que recorrerse todo el contenedor. Dado las limitaciones con la gestión de hilos con TBB y OpenMP el hilo padre no puede continuar la ejecución hasta que todos los demás hilos terminen, esto provoca que en el caso de que por ejemplo el primer hilo encuentre al comienzo del primer sector una coincidencia, el hilo padre tiene que esperar hasta que terminen todos los hilos para saber el primer iterador que contiene el valor buscado. En cambio con ISO Threads en cuanto el primer hilo encontrase el resultado, el padre sabe el resultado y puede seguir la ejecución sin esperar a los demás hilos y devolver el resultado mucho antes.

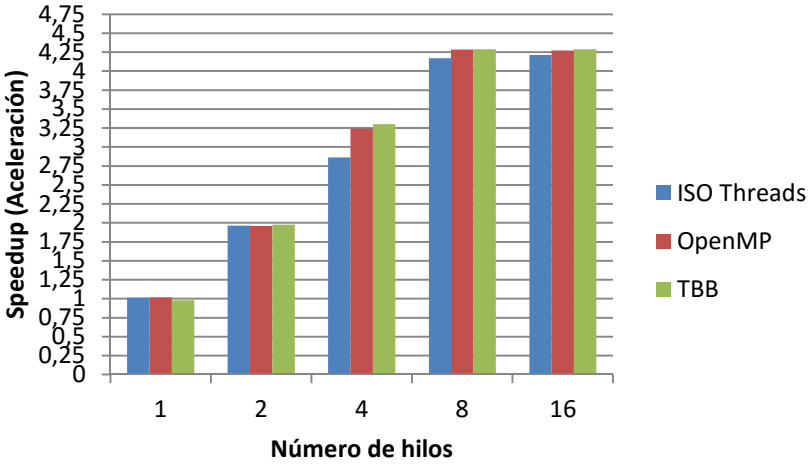



Ilustración 7: Gráfico de aceleraciones de Find

### 6.1.3 Copy

El gráfico (ilustración 8) contiene resultados muy parecidos a los anteriores algoritmos en los que en estos casos destaca OpenMP, pero de todas formas todas las versiones llegan a unos speedups bastante elevados y similares.

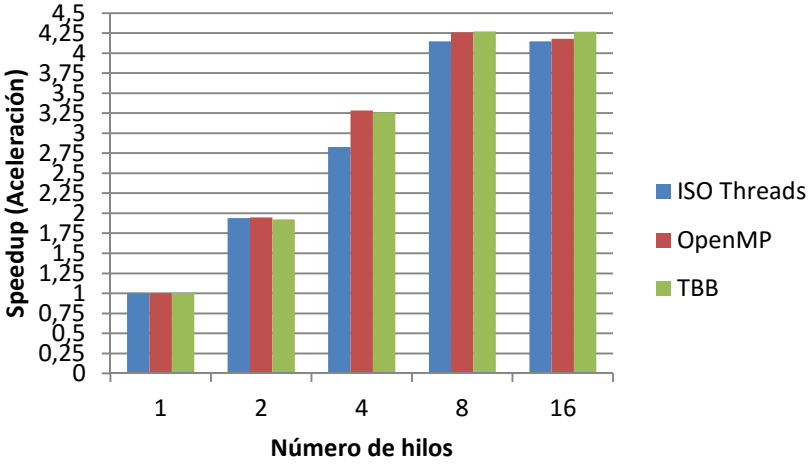



Ilustración 8: Gráfico de aceleraciones de Copy

### 6.1.4 Transform

El siguiente gráfico (ilustración 9) muestra unos speedups bastante elevados y similares para todas las versiones, en este destaca levemente OpenMP en todos los caso pero no con mucha diferencia. Vuelve a suceder que ISO Threads con 4 hilos no da buenos resultados y siempre se queda un poco descolgado.

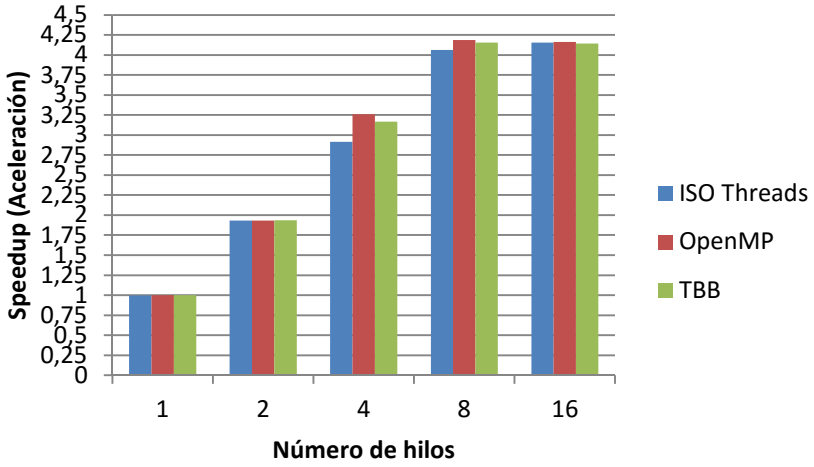



Ilustración 9: Gráfico de aceleraciones de Transform

### 6.1.5 Move

Analizando el gráfico (Ilustración 10) se vuelve a tener la misma distribución que en los primeros casos, speedups muy similares destacando OpenMP y TBB por encima de ISO Threads pero no con mucha diferencia.

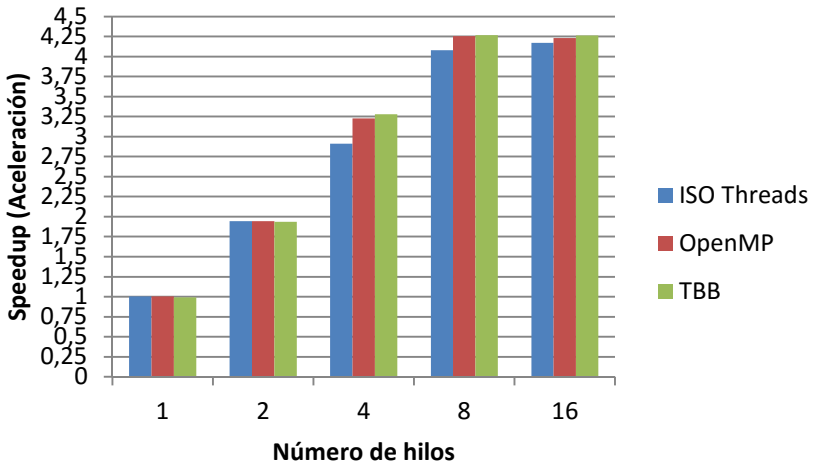



Ilustración 10: Gráfico de aceleraciones de Move

### 6.1.6 Fill

Observando la distribución del gráfico (Ilustración 11) se tiene algo muy parejo a los anteriores, donde la versión algo más sobresaliente es TBB pero sin mucha diferencia.

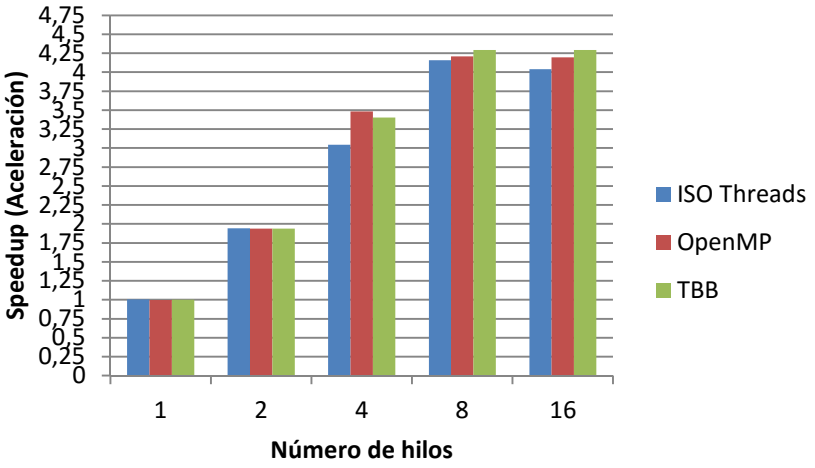



Ilustración 11: Gráfico de aceleraciones de Fill

### 6.1.7 Generate

Examinando el siguiente gráfico (ilustración 12), a diferencia de los anteriores ISO Threads destaca ligeramente sobre las otras versiones en el caso de 16 hilos e igualando los mejores resultados de las otras dos versiones con 8 hilos.

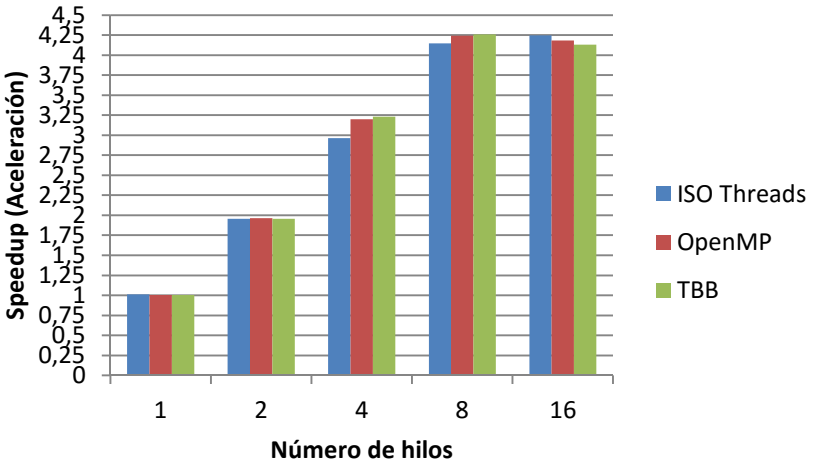



Ilustración 12: Gráfico de aceleraciones de Generate

### 6.1.8 Swap_ranges

El gráfico (Ilustración 13) muestra resultados similares al resto de algoritmos destacando ligeramente TBB.

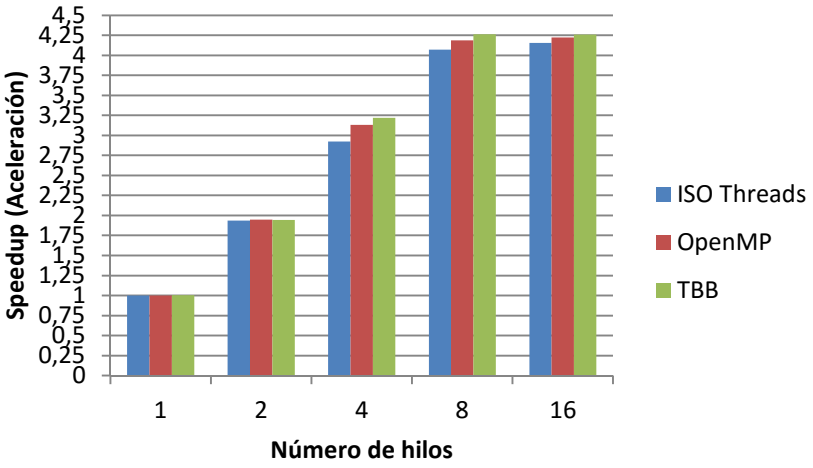



Ilustración 13: Gráfico de aceleraciones de Swap_ranges

### 6.1.9 Accumulate

Los resultados que contiene el siguiente gráfico (Ilustración 14) son muy similares a los que contienen los anteriores destacando OpenMP ligeramente.

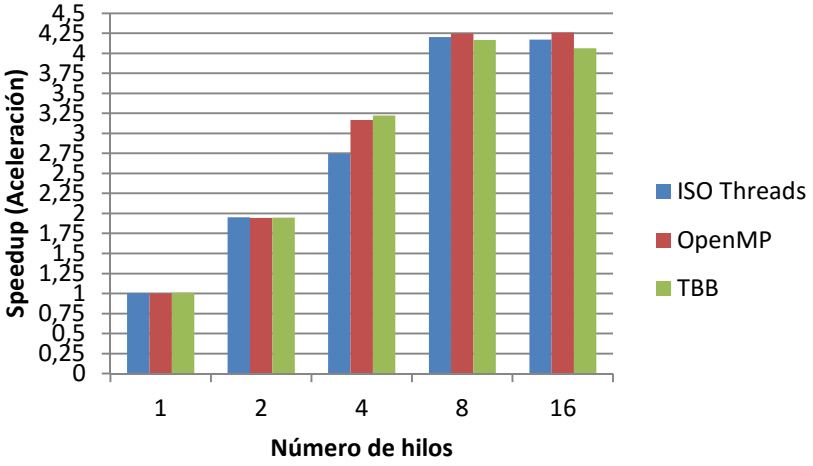



Ilustración 14: Gráfico de aceleraciones de Accumulate

### 6.1.10 Inner_product

Analizando los resultados que muestra el siguiente gráfico (Ilustración 15), los speedups son muy similares al anterior algoritmo, algo normal debido a que la forma de la que está programada es muy similar.

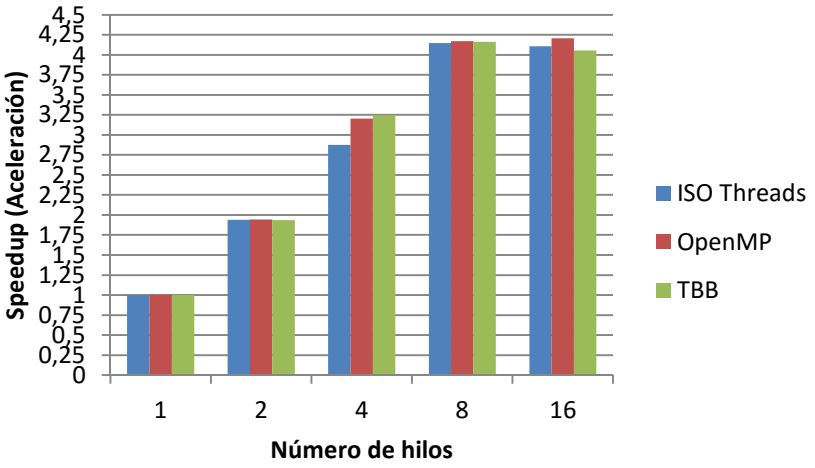



Ilustración 15: Gráfico de aceleraciones de Inner_product

Una vez analizados todos los gráficos y visto los speedups conseguidos para cada algoritmo, se puede decir que estos ha sido bastante buenos y elevados. Por lo tanto las técnicas de paralelización han sido las correctas y podrían usarse en casos reales. Lo más llamativo ha sido como ISO Threads con 4 hilos obtiene siempre unos resultados considerablemente peores respecto a las otras versiones, por lo tanto era una cuestión a analizar y se ha encontrado el motivo. La toma de medidas de estos algoritmos ha sido realizada haciendo la media de 100 ejecuciones para cada uno de los algoritmos, versiones y número de hilos. Estas ejecuciones en el peor caso que es el secuencial rondaban los 16 segundos, por lo tanto esta diferencia de speedup viene dada por dos motivos, ISO Threads con 4 hilos para estos benchmarks nos ha dado resultados menos estables que el resto de las versiones, por lo tanto el porcentaje de variación que puede tener la media es mayor, si a esto le sumas que las ejecuciones más largas son de 16 segundos y las más cortas han sido de entorno 3 segundos, los speedups varían bastante con poca variación de tiempo. Esto hace que si la media tiene más variación de error y ese pequeño error influye mucho en los speedups pueden darse resultados como los obtenidos, donde se obtiene unos speedups inferiores a lo normal.

## 6.2 Evaluación del caso práctico

Dado los resultados obtenidos en los microbenchmarks, se procede a analizar los speedups conseguidos por el algoritmo *transform* paralelizado en la aplicación Blackscholes modernizada. De esta hay de dos versiones como ya se ha comentado, por lo tanto se analizarán de forma separada. Estas pruebas han sido realizadas con la entrada nativa del benchmarks ya que es la que más carga nos ofrece.

### 6.2.1 Resultados de la versión estructura de vectores

Para saber más o menos si el speedup conseguido es bueno o malo primeramente hay que calcular el máximo speedup teórico por la ley de Amdahl. Esta aplicación tiene tres partes, lectura de datos, cómputo y escritura de datos. La única parte que se ha parelizado ha sido la de computo por medio de *transform*, por lo tanto es necesario saber qué porcentaje del tiempo lleva la parte secuencial y la que se puede paralelizar.

Ejecutando la versión secuencial y añadiendo trazas de tiempos en cada sección y tomando tiempos con el comando perf, se obtiene que la fase secuencial compuesta por lectura y escritura es de 36.028 segundos y el total de la ejecución 181,325. Por lo tanto la parte secuencial lleva el 19.869 % del tiempo y la paralela el restante, un 80.131%.

Sustituyendo en la fórmula de la ley de Amdahl se obtiene la aceleración máxima teórica, donde N es el número de núcleos del ordenador de prueba, que son 8 núcleos.

$$Aceleración\ máxima = \frac{1}{(1 - 0.80131) + \frac{0.80131}{8}} = 3.3461$$

A continuación la gráfica (Ilustración 16) contiene es las aceleraciones conseguidas con cada una de las tecnologías y con diferentes números de hilos.

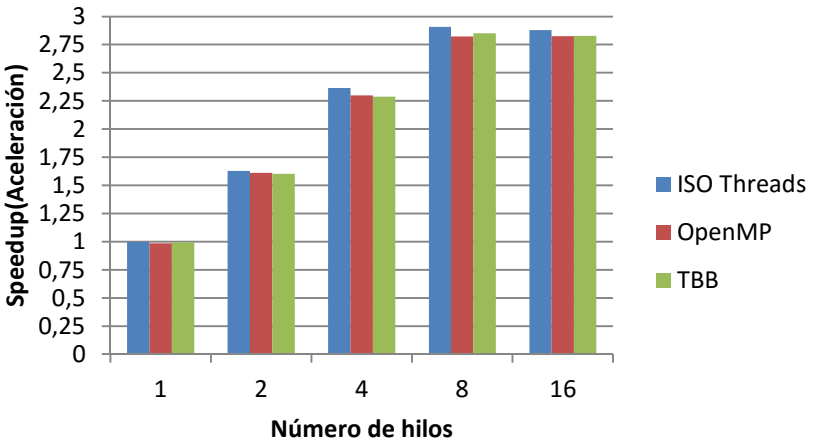



Ilustración 16: Gráfico de aceleraciones Estructura de Vectores

Analizando la gráfica se puede observar como las mayores aceleraciones son conseguidas con 8 y 16 hilos, y destaca por encima de las demás en todos los casos cuando se ha usado ISO Threads. El speedup más alto es cercano a 3, exactamente es 2,9083, lo que se puede interpretar como que el speedup conseguido es muy cercano al máximo teórico y por lo tanto se ha realizado una buena paralelización del algoritmo. La diferencia que se obtiene respecto al máximo puede venir por el tiempo de gestión de los hilos, ya que durante la ejecución se han creado un gran número de hilos y por la gestión de los datos de cada uno.

### 6.2.2 Resultados de la versión vector de estructuras

Con esta versión se procede igual que con la anterior y se obtienen los tiempos de cada una de las partes, en este caso la parte secuencial tarda 36.654 segundos y el total 158,903 segundos. Por lo tanto la parte secuencial supone un 23.0669 % del tiempo y la paralela un 76.9331 %.

Sustituyendo en la fórmula de Amdahl se obtiene la máxima aceleración teórica:

$$Aceleración\ máxima = \frac{1}{(1 - 0.769331 +) + \frac{0.769331}{8}} = 3.059$$

A continuación la gráfica (Ilustración 17) muestra las aceleraciones conseguidas con cada una de las tecnologías y con diferentes números de hilos.

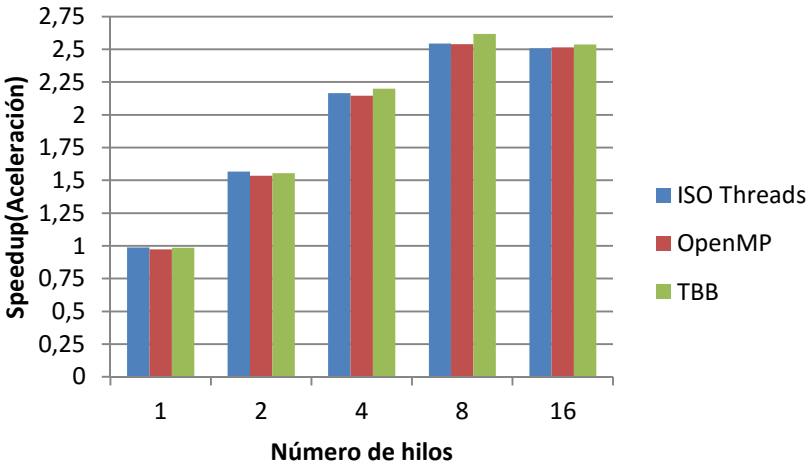



Ilustración 17: Gráfico de aceleraciones Vector de estructuras

Analizando la gráfica se observa cómo se han obtenido unas aceleraciones cercanas a la máxima calculada previamente, en este caso la máxima aceleración se ha conseguido con TBB con 8 hilos, que en este caso ha sido 2.616. La diferencia que se obtiene respecto al máximo puede venir como en el caso anterior por el tiempo de gestión de los hilos y por la gestión de los datos de cada uno.

Observando las dos versiones se puede decir que la paralalelización ha sido exitosa, obteniendo speedups muy cercanos al máximo para todas las versiones. La diferencia existente es normal debido a la gran cantidad de hilos creados, esta gran cantidad de hilos creados es debido a que los cálculos se repiten 100 veces para sobrecargar la aplicación. En aspectos de tiempo ambas versiones han dado buenos resultados, en secuencial la versión de vector de estructuras daba mejores resultados, pero en cambio al realizar la paralelización se han obtenido tiempos muy similares en ambas versiones, esto es debido a que la versión de estructura de vectores tenía más margen de mejora y porque la distribución de la memoria se aprovecha mejor.

Una vez realizadas todas las medidas tanto del caso práctico y de los microbenchmarks, se pueden comparar los resultados de ambos respecto al algoritmo *transform*. En el microbenchmark destacaba ligeramente OpenMP, sin embargo, en el caso práctico en un versión destaca ISO Threads y en otra TBB. Respecto al problema de los microbenchmarks con ISO Threads y 4 hilos en el caso práctico no ocurren, de hecho ISO Threads destaca ligeramente con ese número. En el caso práctico esta versión del algoritmo ha sido muy estable al igual que las demás, esto sumado a que el tiempo de ejecución de la aplicación es más elevado ha hecho que no ocurran los problemas que había con los microbenchmarks.

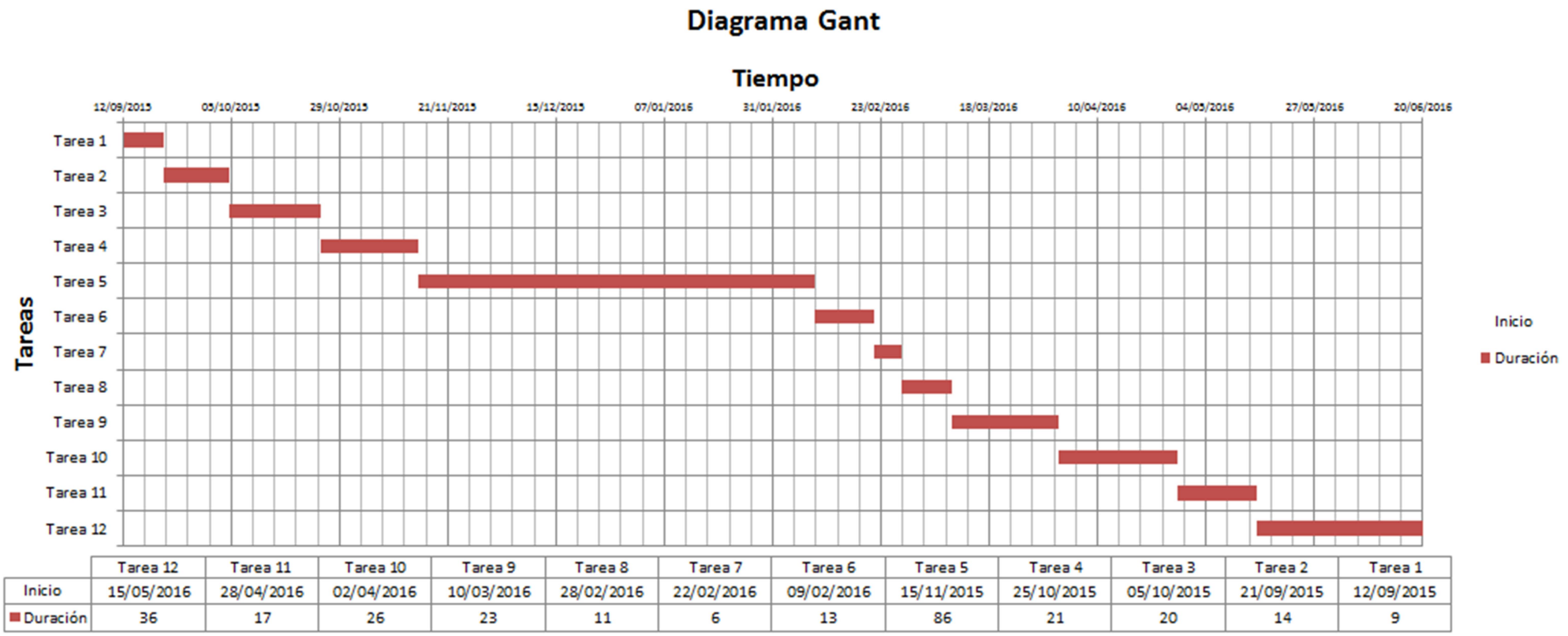
Por lo tanto a la hora de elegir una tecnología de paralelización hay que ver varios factores, las variaciones de tiempo que se obtienen entre varias ejecuciones de una misma versión, los speedups que se consiguen y cuanto varían los speedups con variaciones de tiempo. Además de que no siempre va ser mejor una tecnología, solo hay que ver como hay diferencia entre la tecnología que más speedup ha ofrecido en el microbenchmark de *transform* y en el caso práctico o incluso entre varias versiones de la misma aplicación como en el caso práctico.

# 7. Planificación del trabajo

Con objetivo de una organización clara en la ejecución del Trabajo Fin de Grado y la estimación de plazos, recursos y presupuesto, se establecieron las tareas a realizar para una gestión de ellas lo más eficiente posible. En la siguiente tabla se pueden observar todas las tareas que se establecieron y seguidamente un diagrama Gantt (Ilustración 18) en el que se pueden observar las fechas de comienzo de cada tarea y cuantos días supusieron.

| Identificador | Tarea | Descripción | Tiempo estimado | Tiempo real |
|---|---|---|---|---|
| **1** | **Instalación y configuración del entorno de trabajo** | Preparación del computador de desarrollo (instalación del SO, compilador, instalación de TBB, etc). | 10 horas | 10 horas |
| **2** | **Instalación de Benchmark PARSEC** | Instalar el Benchmark PARSEC y probar todas las aplicaciones en el computador de desarrollo. | 10 horas | 10 horas |
| **3** | **Evaluación aplicaciones de Benchmark PARSEC** | Análisis de aplicaciones con objetivo de elegir una aplicación para el caso práctico. | 20 horas | 40 horas |
| **4** | **Establecer entorno de compilación para la aplicación seleccionada** | Establecer una estructura de carpetas organizadas y configurar CMake para construir MakeFile para la compilación. | 10 horas | 15 horas |
| **5** | **Modernización de la aplicación seleccionada** | Cambiar el código de la aplicación a C++, modernizando el código con objetivo de poder utilizar la biblioteca de algoritmos. | 60 horas | 80 horas |
| **6** | **Elaboración versión secuencial de los algoritmos.** | Elegir los algoritmos a analizar y realizar su programación de forma secuencial. | 30 horas | 30 horas |
| **7** | **Elaboración de la versión Pthreads de los algoritmos** | Programar la versión paralela de los algoritmos con la tecnología ISO Threads. | 20 horas | 15 horas |
| **8** | **Elaboración de la versión OpenMP de los algoritmos** | Programar la versión paralela de los algoritmos con la tecnología OpenMP. | 20 horas | 25 horas |
| **9** | **Elaboración de la versión TBB de los algoritmos** | Programar la versión paralela de los algoritmos con la tecnología TBB. | 30 horas | 30 horas |
| **10** | **Elaboración de microbenchmarks** | Programas pequeños programas sencillos donde probar y medir rendimientos de los algoritmos. | 20 horas | 25 horas |
| **11** | **Toma de tiempos definitivos** | Tomar los tiempos en los microbenchmarks y en la aplicación del Benchmark PARSEC modernizada. | 30 horas | 40 horas |
| **12** | **Redacción de la memoria del TFG** | Documentación del trabajo realizado. | 40 horas | 50 horas |
| | | **Total de tiempos** | 300 horas | 370 horas |

Tabla 26: Planificación del Trabajo Fin de Grado




| | Tarea 12 | Tarea 11 | Tarea 10 | Tarea 9 | Tarea 8 | Tarea 7 | Tarea 6 | Tarea 5 | Tarea 4 | Tarea 3 | Tarea 2 | Tarea 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Inicio | 15/05/2016 | 28/04/2016 | 02/04/2016 | 10/03/2016 | 28/02/2016 | 22/02/2016 | 09/02/2016 | 15/11/2015 | 25/10/2015 | 05/10/2015 | 21/09/2015 | 12/09/2015 |
| Duración | 36 | 17 | 26 | 23 | 11 | 6 | 13 | 86 | 21 | 20 | 14 | 9 |

Ilustración 18: Diagrama de Gantt

# 8. Presupuesto

En este apartado se expone el presupuesto calculado para la realización de este Trabajo Fin de Grado. Para el cálculo se ha tenido en cuenta las horas que ha llevado cada tarea que son 370 horas y teniendo en cuenta el tiempo que ha llevado el proyecto que serían 282 días, en torno a 9 meses. Dado que el proyecto ha sido desarrollado con fines académicos y de a investigación no se han aplicado porcentajes extra de riesgos y beneficios en los costes del personal.

Los costes de personal pueden verse en la siguiente tabla:

| Puesto | Horas totales | Coste/hora | Coste total |
|---|---|---|---|
| Jefe de proyecto | 80 | 52€ | 4.160€ |
| Ingeniero de desarrolló | 370 | 24€ | 8.880€ |
| **Coste: recursos humanos** | | | **13.040€** |

Tabla 27: Coste de recursos humanos

Los costes de software y hardware pueden verse en la siguiente tabla:

| Elemento | Coste unitario | Tiempo de vida | Coste mensual | Meses utilizado | Coste amortizado |
|---|---|---|---|---|---|
| **Hardware** | | | | | |
| MSI Cx61 | 800€ | 36 | 22,22€ | 9 | 200€ |
| Toshiba L50-C-200 | 700€ | 36 | 19,44€ | 9 | 175€ |
| **Coste hardware** | | | | | **375€** |
| **Software** | | | | | |
| Licencia Microsoft Office 2010 | 199,00€ | 50 | 4€ | 9 | 36€ |
| Software libre | 0,00€ | 50 | 0€ | 9 | 0 |
| **Coste software** | | | | | **36€** |
| **Coste: hardware y software** | | | | | **411€** |

Tabla 28: Coste de hardware y software

Por ultimo en la siguiente tabla se puede ver el coste final de este Trabajo Fin de Grado:

| Elemento | Coste total |
|---|---|
| Recursos humanos | 13.040€ |
| Hardware y software | 411€ |
| **Coste total** | **13.451€** |

Tabla 29: Coste total de Trabajo Fin de Grado

# 9. Marco legal y normativo

En este apartado se hace referencia al aspecto legal del Trabajo Fin de Grado. Este marco legal está compuesto por los estándares del lenguaje C++, las licencias de las tecnologías utilizadas para la paralelización y la licencia del benchmark PARSEC.

## 9.1 Lenguaje C++

**ISO/IEC 14882:2011 y la revisión de este que es ISO/IEC 14882:2014** especifican los requisitos de programación del lenguaje C++. El primer requisito es que ellos implementan el lenguaje y por lo tanto ISO/IEC 14882:2011 también define C++. Y especifica que C++ es un lenguaje de programación basado en el lenguaje de programación C como se describe en el estándar ISO/IEC 9899:1999. Además de las facilidades añadidas por C, C++ proporciona tipos de datos adicionales, clases, plantillas, excepciones, espacios de nombres, sobrecarga de operadores, sobrecarga de funciones, referencias, funciones de liberación de memoria y servicios adicionales de biblioteca.

## 9.2 Tecnologías utilizadas

ISO Threads: Fue definido por primera vez en el estándar ISO/IEC 14882:2011 del que se ha hablado anteriormente.

OpenMP: **Copyright © 1997-2015 OpenMP Arquitectura Review Board.** El que se concede permiso para copiar sin cargo todas o parte de este material, proporcionado el aviso de copyright de OpenMP Architecture Review Board y el título de este en el documento.

TBB: tiene licencia dual, licencia comercial (COM) y GPLv2[18] basada en licencia de código abierto (OSS). Por lo tanto como nuestro uso es académico hay que tener en cuenta la licencia de código abierto.

## 9.3 Benchamark PARSEC

El Benchamark PARSEC tiene **Copyright © 2006-2009 Princeton University** donde la redistribución y uso está permitido siempre que se cumplan las siguientes condiciones:

- Las redistribuciones del código fuente deben conservar el aviso de copyright anterior, esta lista de condiciones y la siguiente renuncia de responsabilidad.
- Las redistribuciones en formato binario deben reproducir el aviso de copyright anterior, esta lista de condiciones y la renuncia siguiente en la documentación y/u otros materiales suministrados con la distribución.
- Ni el nombre de la Universidad de Princeton ni el nombre de sus colaboradores puede utilizarse para respaldar o promover productos derivados de este software sin permiso escrito específico.

Renuncia de responsabilidad:

"THIS SOFTWARE IS PROVIDED BY PRINCETON UNIVERSITY ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PRINCETON UNIVERSITY BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE."

# 10. Conclusiones

## 10.1 Verificación de requisitos

### *REQF-1: Contenido de la biblioteca*

La biblioteca está formada por los algoritmos count, find, copy, transform, move, fill, generate, swap_ranges, accumulate, inner_product, estos completan 10 algoritmos que pertenecen a diferentes familias.

### *REQF-2: Versión secuencial de los algoritmos*

La biblioteca contiene una versión secuencial para todos los algoritmos que no hace referencia al algoritmo original de la STL.

### *REQF-3: Versión paralela de los algoritmos con ISO Threads*

La biblioteca contiene una versión paralela para todos los algoritmos implementada con ISO Threads

### *REQF-4 Versión paralela de los algoritmos con OpenMP*

La biblioteca contiene una versión paralela para todos los algoritmos implementada con OpenMP.

### *REQF-5: Versión paralela de los algoritmos con TBB*

La biblioteca contiene una versión paralela para todos los algoritmos implementada con TBB.

### *REQF-6: Mismos argumentos más los extras*

Todos los algoritmos reciben por argumento primero la política, luego los argumentos originales que recibían y por último el número de hilos con el que es ejecutado el algoritmo.

### *REQF-7: Definición condicional de las interfaces para los tipos de versiones*

Al utilizar enable_if en el código ejecutable solo aparecerán las versiones utilizadas, ya que se comprueba que política es para saber que código hacer efectivo en tiempo de compilación.

### *REQNF-1: Implementación, pruebas y análisis en Linux*

Todo el desarrollo del proyecto ha sido realizado en el SO Linux.

### *REQNF-2: Microbenchmarks para cada algoritmo*

Todos los algoritmos han sido probados y analizados sus rendimientos con microbenchmarks previamente desarrollados.

### *REQNF-3: Evaluar aplicaciones de PARSEC y elegir una*

Se evaluaron todas las aplicaciones y la elegida fue blackscholes dado a su simplicidad, el problema que resuelve y que se podía utilizar el algoritmo *transform*.

### *REQNF-4: Evaluar un algoritmo en un caso práctico*

El algoritmo *transform* fue sometido a pruebas en la aplicación Blackscholes y se evaluaron los rendimientos de la aplicación respecto a este algoritmo.

### *REQNF-5: Modernización del código*

Blackscholes estaba implementada originalmente en lenguaje C y se realizaron dos modernizaciones del código en lenguaje C ++ para poner a prueba el algoritmo *transform*.

### *REQNF-6: Compilación con GCC*

La compilación siempre ha sido realizada con GCC y el entorno de compilación fue creado a partir de Cmake.

## 10.2 Líneas futuras

Como próximos trabajos futuros, propondría la continuación de la biblioteca implementada para todos los algoritmos de la STL, incluyendo además en el futuro dos políticas extras, que corresponderían a hacer llamadas a los algoritmos paralelos de C++17 con las dos políticas que disponen, con esto se conseguiría que el usuario con el cambio de un argumento de la función pueda realizar pruebas con un número considerable de tecnologías de paralelismo y de concurrencia.

En lo referente a blackscholes una buena idea sería continuar con la modernización de dicho programa, con más tiempo y detenimiento. Utilizando C++ se podría conseguir que la ejecución del programa fuese más eficiente que lo conseguido hasta ahora con C. Además de intentar paralelizar la aplicación con otras tecnologías como Intel ArBB o CUDA.

Por último y volviendo a la biblioteca implementada, también propondría medir rendimientos en casos prácticos de alta carga computacional, en los que puedan ser utilizados cada uno de los algoritmos y ponerlos a pruebas en computadores mucho más potentes que los disponibles en este Trabajo Fin de Grado.

## 10.3 Conclusiones técnicas

Los resultados han sido más o menos los esperados, unos speedups (aceleraciones) muy considerables en los microbenchmarks, en los casos con más hilos siempre por encima de 4 y en el caso práctico se consiguió acercarnos mucho al máximo de aceleración teórico según la ley de Amdahl. Con los resultados del caso práctico podemos afirmar que en situaciones reales y con gran volumen de datos estos algoritmos paralelos pueden ser de gran ayuda sin un coste de líneas de código. Además estos resultados nos han indicado un aspecto que hay que tener muy en cuenta al elegir que versión de paralelización usar, que es la variación en el tiempo de ejecución entre varias ejecuciones, porque no es lo mismo utilizar una versión que a

veces da muy buenos resultados y a veces no tanto, que otra que es más estable y da siempre buenos resultados. Esta estabilidad de las versiones de los algoritmos depende de donde se use por lo tanto en aspectos de desarrollo siempre hay que tomar muchas tomas de tiempo para poder elegir.

Hay que añadir que hay algoritmos que solo los recomiendo usar cuando el tipo de dato sobre el que se trabaja no es de tipo básico, sino en casos donde las operaciones por ejemplo de comparación o asignación tienen una carga computacional moderada o alta.

## 10.4 Conclusión personal

Como conclusión final quiero manifestar mi opinión personal respecto el Trabajo Fin de Grado. La biblioteca desarrollada me parece de gran utilidad, sobre todo porque aísla de la programación paralela al usuario y le da la facilidad de que sin saber nada de paralelización puede paralelizar algo con varios mecanismos. Muy útil sobre todo en el momento en el que vivimos que cualquier persona programa y no tienen conocimientos de lo que es la paralización o por ejemplo un hilo de ejecución.

Los algoritmos de la STL son muy utilizados y era necesario que tuviesen versiones paralelizadas, de ahí surgió la idea de este Trabajo Fin de Grado, también teniendo en cuenta que en la revisión de C++ en 2017 se hará algo parecido a lo que hace nuestra biblioteca, como ya se ha comentado en apartados anteriores. Ha sido una pena no poder realizar la biblioteca con todos los algoritmos, pero el gran número algoritmos y la cantidad de pruebas y comprobaciones que hay que hacer nos ha obligado a hacerlo de un número representativo.

Como opinión personal creo que bibliotecas de este tipo deben ir apareciendo en todos los lenguajes de programación, no solo en C++, sobre todo en estos momentos donde cualquier programa tiene que recurrir al uso de hilos para cumplir los objetivos.

Respecto a las tecnologías utilizadas para la paralelización, para mí la más completa y ofreciendo un gran rendimiento es ISO Threads, también es verdad que los conocimientos para una buena paralelización con dicha tecnología no son muy comunes y la implementación es mucho más costosa. Por otro lado es verdad que las facilidades que dan OpenMP y TBB para paralelizar bucles por ejemplo, son muy sencillas y eficientes, pero les falta esa gestión de bajo nivel que marca la diferencia en cuanto a limitaciones en casos concretos en los que es necesario mucho control respecto a los hilos.

En cuanto a mis objetivos que tenía con este Trabajo Fin de Grado, he conseguido lo que tenía pensado y además me llevo mucho más conocimiento añadido del que pensaba. Primeramente he aprendido a utilizar herramientas de control de versiones, he adquirido cierto dominio sobre el lenguaje C++ que era mi principal objetivo, he

aprendido mecanismo de paralelismo como TBB con el que no había trabajado antes y he conseguido afianzar mis conocimientos sobre ISO Threads y OpenMP. Por otro lado también he aprendido a optimizar código viendo ciertas características de ejecución como fallos cache de datos, ciclos de reloj, fallos de cache de instrucciones, etc. Aspectos que en la carrera he tratado teóricamente pero que nunca había usado en un caso práctico, dándome cuenta que sin una base hardware de calidad es inviable desarrollar un programa que tenga un gran rendimiento.

# 11. Summary in English

## 11.1 Introduction

This first section explains the general vision of this work order degree, the motivation of this work and the preliminary objectives.

### 11.1.1 Motivation

During the last years, one of the major objectives to get by the engineers has been increase the performance in the processors. Firstly they focused in take advantage of processors clock frequency with objective of improving its performance, to do that the number of transistors was increasing. Moore formulated his law in the year 1965 and it was rectified in 1975, this law affirmed that the number of transistor of the processors would increase doubling his number each 24 month. This increase has been fulfilled until very recently, thanks to reduce the size of the logic gates, but this increase has been slowed because of physical limits of the semiconductors. These physical limits have caused problem with consumption, heat dissipation and the necessity of synchronized information. Then, vector instructions were introduced getting a best performance in these sequential processors due to use the SIMD (Single Instruction Multiple Data) model. As a result the data-level parallelism was achieved that consists of applying the operation of one instruction on a large data set. This improvement was not enough and this architecture was replaced by new generations of processors.

The illustration 1 shows how the clock frequency processors, its consumption and the data-level parallelism reached its limit more than a decade ago. That's why the "multi-cores" revolution came. This processors mix several processors in the same integrated circuit. Thanks to this several programs can run on different cores at the same time, or what is better, the same program can run in several cores. The thread level parallelism is necessary to exploit this architecture, these threads are light processes that the operative system scheduler. The threads are created by a father process that he can create as many threads as the multi-core supports, so several lines of code of the same program can be executed concurrent.

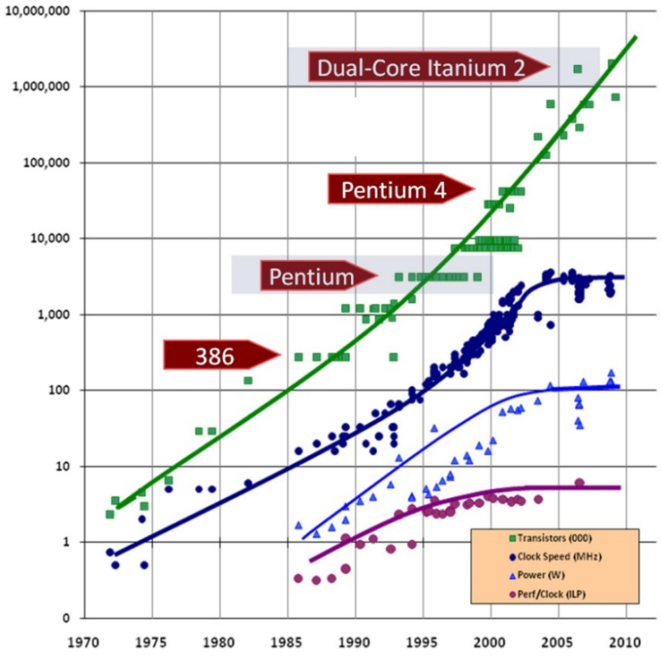



Illustration 19: Trend of processors according to Moore's law

The parallel programming is very complicated, is certain that the apparition of library of function that extend of sequence languages as POSIX Threads or ISO Threads, or the parallel languages, as OpenMP or TBB have eased the parallel program. But these don't prevent that the programmer doesn't have to worry about the scalability problem, the synchronization between threads or generated locks.

Given this complexity of parallel programming, arise the idea of creating a parallel library with simple algorithms that are regularly used in programming. These algorithms are those that make up the STL of C++ and can be used easily for each of the containers that STL provides. These containers are vectors, maps, list…, but also user-created containers use these algorithms that complete the STL structure. This is possible because the programming of this algorithm is generic, so they are independent of the type of data. With this parallel library the programmer would forget the problems of parallelization and could make use of these as If it were the sequential version.

In addition, I must emphasize my personal interest for doing a project in C++ because this language is one of most used in the programming and is hardly used during the college career. Besides the interest to know some techniques of parallelism that can be used.

### 11.1.2 Objectives

The main objective of this work order degree is the creation of a library that has parallelized the algorithms of C++ STL. Due to the number of algorithms is very high, the library will have a representative number of them to demonstrate the viability of the parallelization of all. These algorithms will be parallelized with three technologies: ISO Threads, OpenMP and TBB. The library will provide facilities to change the technology with few changes. Also, this work order degree will do an analysis of the speedup achieved with the parallelization of the algorithms in relation with her sequential version. Likewise, this has the following secondary objectives:

- Evaluate the different applications of the suite benchmark PARSEC in order to discover the utility of benchmarks and choosing a real application where can make use of our library.
- Upgrade the benchmark application code chose, in our conversion from a pure code C to C++.
- Implement small benchmarks for testing each algorithm.
- Show a global vision of parallels architectures with shared memory, emphasizing in ISO Threads, OpenMP and TBB.
- Check if there are big performance differences between OpenMp, TBB or ISO Threads.

## 11.2 Evaluate of results

The performance evaluation of the programing algorithms is made in two ways, with micro benchmarks, which are simple programs previously developed with the objective of checking the performance of all algorithms, and with a case study, that is one application of benchmarks PARSEC where one algorithm is used and evaluated their performance.

The benchmarks don't give reliability 100% on the results, i.e., if a benchmark show that particular version of the algorithms is the best, doesn't have to be true for all cases, that is depending on the problem, the same version could be the best and the worst in different situations.

The tests have been made in a computer with processor Intel i7-472HQ, with four physical cores with 2.3 Ghz e hyper-threading, o what is the same, eight logical cores. In addition, this computer has 8GB of memory RAM. The captures of time have been through with the linux kernel perf that also shows the hardware counters of an execution.

### 11.2.1 Microbenchmarks evaluation

The little programs have been created to evaluate the performance of the algorithms, as operations with basic types that are very quick and easily optimized by the compiler. The tests have been done about a created class that has an int and a float and has overloaded the following operators:

- Sum: Where adding two object of the same class. On the one hand the int is added, and the other the float is added.
- Equality: If the two attributes of classes are equals the result will be true.
- Allocation: A new object obtains value from another giving value to his attributes.

All the operators less the sum have computing overload to simulate that the operator takes more execution time. In addition, it is necessary that the class has the constructors to initialise the objects.

The following two graphics show the results of the algorithms *transform* and *move*. The speedups achieved are very high, in some cases getting more threads above four of speedup and obtaining very good speedups with few threads. All algorithms have generated very similar graphics, which the differences are very small and these get that the best parallelization version changes for each algorithm. Something to note is that with 4 threads ISO Threads provided speedups more lower than the other versions in all cases.

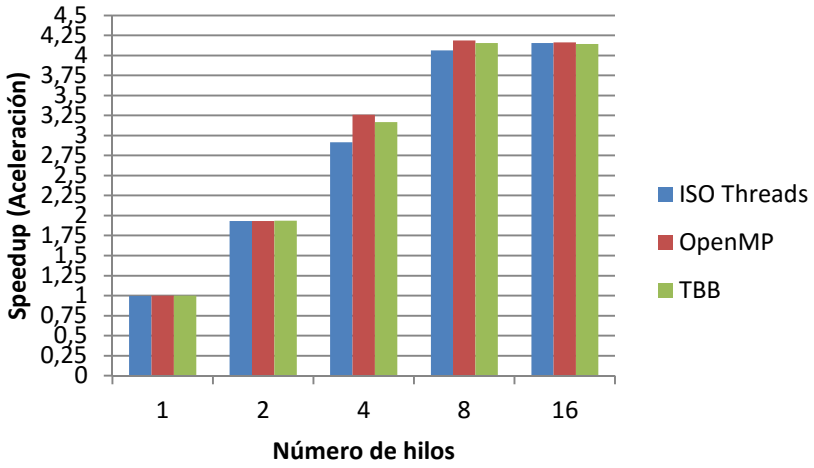



Illustration 20: Graphic of speedups of Transform

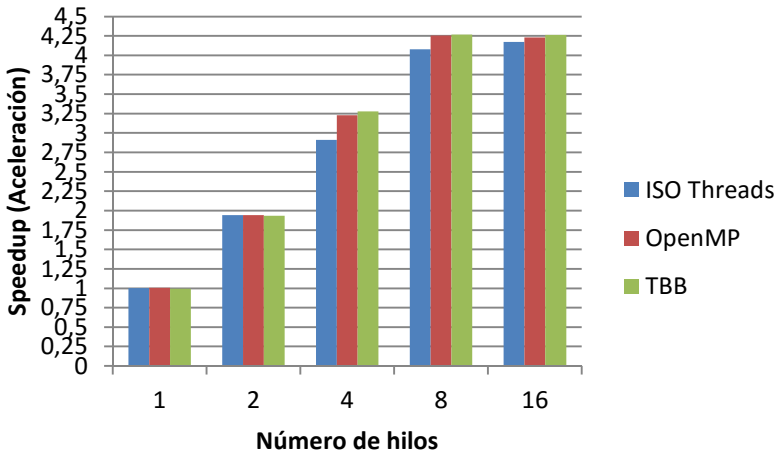



Illustration 21: Graphic of speedups of Move

These good results assure that the technical parallelization has been the correct and these parallel algorithms can be used in real applications. The most surprising is that ISO Threads, with four threads, have given a result considerably worse to other versions. So, it was interesting to know the reason. The captures of time have been done with the average of one hundred of executions for each algorithm, version and number of threads. This execution took sixteen seconds in the worst case, that is the sequential version, and these differences of speedups are for two reasons. The first is that ISO Threads, for these microbenchmarks, has given less stable results than the rest of the versions. Therefore, the percentage of variation that can have the average is higher. The second is that the longest executions are of sixteen seconds and the shortest executions are of three seconds, so the speedups vary quite a lot with little variations of time. Because of the combination of these reasons, results such as those can be obtained.

## 11.3 Evaluate study case

The microbenchmarks have demonstrated that is possible to use these algorithms in real cases, so we will test the Transform algorithm in the application Blackscholes of benchmarks PARSEC. This application has been modernized for the use of the algorithm and has two versions, so it will be analysed separately.

### *Results with the version of structure of vectors*

To find out if the speedup achieved in the application is good or bad, it is necessary to calculate the maximum theoretical by the law of Amdahl. This application has three phases: reading data, computing and writing data. The only phase that has been

paralyzed is the computing phase through Transform, so it is necessary to know that time consums the sequential part and the parallel part.

As a result of running the sequential version, with traces of time in the different sections, the sequential part takes 36.028 seconds and the complete execution takes 181.325 seconds. So the sequential takes the 19.869 % of the time and the parallel 80.131%.

Substituting in the formula has the theoretical maximum speedup, where N is the number of cores in the test computer, which are 8 cores.

$$\text{Maximum speedup} = \frac{1}{(1 - 0.80131) + \frac{0.80131}{8}} = 3.3461$$

The graph (Figure 16) shows the speedups obtained with each of the technologies and different numbers of threads.

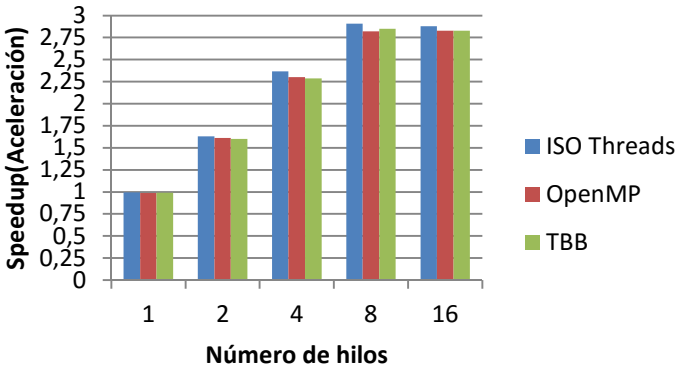



Illustration 21: Graphic of speedups of structure of vectors

Analysing the graph shown, the higher speedups are achieved with eight and six threads and stand out above the others in all case when ISO Threads is used. The high speedup is close to 3, exactly is 2.9083. In conclusion the result is very close to the maximum and therefore the parallelization of the algorithm is good. This difference with the maximum can be for the time management of threads, due to the large number of them and by the data management of each one.

### *Results with the version of vector of structures*

As in the previous version of each of the parts are obtained that the sequential part takes 36.0654 seconds and the total takes 158.90 seconds. So the sequential part takes a 23.0669% of the time and the parallel takes the 76.9331%.

Substituting in the law of Amdahl to get the maximum speedup:

$$\text{Maximum speedup} = \frac{1}{(1 - 0.769331 +) + \frac{0.769331}{8}} = 3.059$$

The following graph (Figure 17) shows the speedups obtained with each of the technologies and the different numbers of threads.

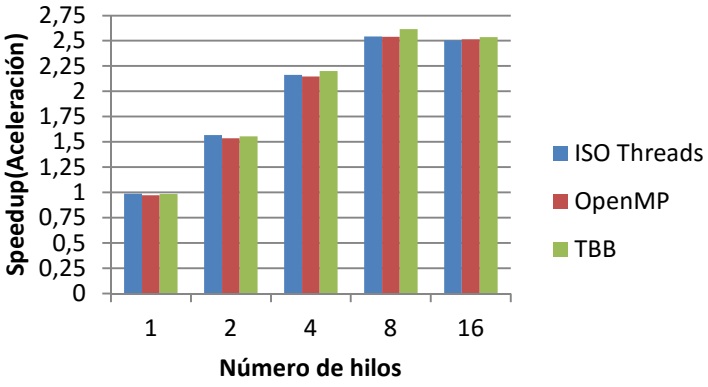



Illustration 22: Graphic of speedups of vector of structures

The graph shows how the speedups obtained close to the previous calculated maximum. In this case the maximum speedup is with TTB with 8 threads, which is 2.616. The difference obtained regarding the maximum speedup possible, as in the previous case, is consequence of the time management of threads due to the large number of them and by the data management of each one.

The analysis of the two versions says that the parallelization has been satisfactory, achieving speedups near the maximum on all the versions of parallelization. The maximum difference is normal because the running program creates lots of threads because the calculations are repeated a hundred times to overload the programme. Regarding the time, both versions have given successful results, in sequential the version of vector of structure gave better results, but in parallel both have similar times. This is because the version of structure of vectors has more margin for improvement and the data distribution is better exploited.

Once the measures of the case study and the microbenchmarks has done, can be compared the result of both regarding the Transform algorithm. In the microbenchmarks excelled OpenMP however in the case study excelled in one version

ISO Threads and in the other version TBB. In relation with the problem of microbenchmarks with ISO Threads and four threads in the study case don't happen, in fact ISO Threads excels with this number of threads in the case study. In the case study, this version of algorithm have been very stable in combination with the higher execution time did that doesn't happen the problem. Therefore to choose a parallelization technology is necessary to see several factors, the changes of time that are obtained between several executions and the speedups obtained.

In addition to that not always a technology is the best, it is necessary only to see as there is difference between the best technology in the microbenchmark of *transform* and the case study or between several versions of the same application.

## 11.4 Conclusions

### 11.4.1 Verification of requirements

#### REQF-1: Content library

The library contains the following algorithms: count, find, copy, transform, move, fill, generate, swap_ranges, accumulate and inner_product. These complete ten algorithms of different families.

#### REQF-2: Sequential version algorithms

The library contains a sequential version for all algorithms which doesn't refer to the original algorithm of the STL.

#### REQF-3: Parallel version of the algorithm with ISO Threads

The library contains a parallel version for each algorithm which is implemented with ISO Threads.

#### REQF-4 Parallel version of the algorithm with OpenMp

The library contains a parallel version for each algorithm which is implemented with OpenMP.

#### REQF-5: Parallel version of the algorithm with TBB

The library contains a parallel version for each algorithm which is implemented with TBB.

#### REQF-6: Same arguments and extras

All the algorithms receive as first argument the policy, after that the origin arguments and the last the number of threads to execute.

#### REQF-7: Conditional definition of the interfaces to the types of updates

Using enable_if the executable of the code only will have the versions used because the compiler checks policy and makes effective the corresponding code.

#### REQNF-1: Implementation, testing and analysis on Linux

All development of the project has been done in Linux OS.

#### REQNF-2: Microbenchmarks for each algorithm

All algorithms have been tested and analysed their performances with previously developed microbenchmarks.

#### REQNF-3: Evaluate applications of PARSEC and choose one

All applications were assessed and the application chosen was blackscholes due to its simplicity, the problem that resolves and that the *transform* algorithm could be used.

#### REQNF-4: Evaluate an algorithm in a case study

The *transform* algorithm was tested in Blackscholes application and the performance of the application was evaluated respect to this algorithm.

#### REQNF-5: Modernization of the Code

Blackscholes was originally implemented in language C and two modernizations of the code were made in language C++ to test the Transform algorithm.

#### REQNF-6: Compile with GCC

The compilation has always been done with GCC and the build environment was created from Cmake.

### 11.4.2 Future lines

As a next future work, I suggest follow the implementation of the library for all STL algorithms, including in the future two policies more, which correspond to the parallel algorithms of STL of C++17 with the two policies that have. So with a simple interface, the users with the change of one argument could testing with a substantial number of technologies of parallelization and concurrency.

Regarding blackscholes, a good idea will be continue with the modernization of this application, with more time and carefully. Using C++ the execution of program could be more efficient than what has been achieved so far with C. Also the application could be parallelized with other technologies such as Intel ArBB o CUDA.

Lastly and returning to the library implemented, also I would suggest measure performance in practical cases with high computational load, where the algorithms of the library can be used and use many more powerful computers than available in this work order degree to take these measure.

### 11.4.3 Technical conclusions

The results have been more or less expected, a speedup very considerable in the microbenchmarks and the speedups in the practice case have been very near to maximum theoretical that has been calculated with the law of Amdahl. The results of the practical case claim that the algorithms can be used in a real situation, with a large data volume and these are of grand help without a cost of lines code.

In addition these results show an aspect that must be taken in account when choosing a version of parallelization, which is the variation in runtime between several executions. That's why it is not the same using a version that gives very good result and occasionally a worse result than one more stable that always gives good results. This stability of versions of algorithms depends on where they are used, therefore in aspect of development always have to take many times measures to choose the version.

I have to add that many of these algorithms only are recommended when the type of data doesn't is a basic type, but in the case where the operation of assignment or compare has a moderate or high computational load.

### 11.4.4 Personal conclusion

As conclusion I want manifest my personal opinion regarding work order degree. I find the library development very useful, especially because this isolates the user of the parallel programing and this give the facility that without know anything about parallelization the user can parallelized something with several mechanics. That is very useful especially in this moment, where anyone program and this doesn't have knowledge about what the parallelization is or what a thread of execution is. The STL algorithms are very used and was necessary that they had a parallel version, hence the idea of this work order degree. Also considering that the revision of C++ in 2017 includes anything similar to our library. It has been a pity not to can do the library with all the algorithms, but the large number of them and the amount of tests and checks to do have forced us to do only a representative number.

As personal opinion I believe that this type of libraries should appear in all programming languages, not only in C++, especially in this moment where all program need to use threads to accomplish the objectives.

In relation to the technologies used for the parallelization, in my opinion, the most complete, without limit and offer a grand performance, is ISO Threads. Also is true that the knowledges necessary to use this technology aren't very common and the implementations are more expensive. However is true that the ease that gives OpenMP and TBB to paralyze loops, for example, are very simples and efficient, but they lack of low-level management makes the difference in terms of limitations.

Regarding my personal goals in this work order degree, I have obtained a lot of knowledge added. Firstly I have learned to use tools of version control, also I have acquired certain mastery over the language C++ that was my principal objective. In addition, I have learned mechanisms of parallelized as TBB which I had never worked with it and I have gotten secure my knowledges about ISO Threads and OpenMP. What's more I have learned to optimize code watching certain of execution features as cache fail, clock cycles, cache of instructions fail, etc. That are aspects that I have used theoretically but I had never used in the practice in the college career. This has made me realize that without a hardware base of quality is impossible development a program that has a great performance.

# 12. Bibliografía

[1] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 2005.

[2] Gottlieb, Allan, Almasi, George S. Highly parallel computing. Redwood City, Benjamin/Cummings, 1989.

[3] D. A. Patterson and J. L. Hennessy. Computer Organization and Design, Fourth edition, ed. Morgan Kaufmann, 2009.

[4] Bjarne Stroustrup. The C++ Programming Language. Fourth edition. Ed Addison-Wesley, 2013

[5] Livermore National Laboratory Lawrence Blaise Barney. POSIX Threads Programming.

[6] ANSI/IEE POSIX 1003.1c. IEEE Standard for Information Technology--Portable Operating System Interface (POSIX®) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language). IEE, October 1996.

[7] ISO/IEC 14882:2011. Information technology -- Programming languages -- C++, ISO, September 2011.

[8] ISO/IEC 9899:2011. Information technology -- Programming languages -- C, ISO, December 2011.

[9] ISO/IEC 14882:2014 Information technology -- Programming languages -- C++, ISO, December 2014.

[10] Anthony Williams. C++ Concurrency in action. First edition, ed Manning, February 2012.

[11] B. Chapman, G. Jost, and R. v. d. Pas. Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press, 2007.

[12] James Reinders. Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism. First edition, ed. O'Reilly, 2007.

[13] ISO/IEC JTC1/SC22. Programming Languages -- Technical Specication for C++ Extensions for Parallelism. International Technical Specication ISO/IEC 19570:2015, ISO, December 2015.

[14] Christian Bienia, Sanjeev Kumar, Jaswidner Pal Singh and Kai Li. Technical. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Princeton University, 2008.

[15] PARSEC Wiki (2016, June). [Online]. Available: http://wiki.cs.princeton.edu/index.php/PARSEC

[16] P. Lamothe Fernández and M. Pérez Somalo. Opciones financieras y productos estructurados, segunda edición ed. McGraw-Hill. 2003

[17] Miguel Ángel Mirás Calvo, Matemáticas en Wall Street: la fórmula de Black-Scholes http://mmiras.webs.uvigo.es/Webpersonal/docusPDF/lugo.pdf

[18] Perf Wiki (2016, June). [Online]. Available: https://perf.wiki.kernel.org/index.php/Main_Page

[19] GNU General Public License, version 2(2016, June). Available: http://www.gnu.org/licenses/gpl-2.0.html