

**PRÁCTICA 1: CONCURRENCIA BASADA EN TAREAS
EN C++11**

DISEÑO DE SOFTWARE DE SISTEMAS
Curso 2015-2016



Grado en Ingeniería Informática
Universidad Carlos III de Madrid

ALONSO MONSALVE, SAÚL:	100 303 517
PRIETO CEPEDA, JAVIER:	100 307 011

ÍNDICE

1	Introducción	4
2	Descripción del código.....	5
2.1	Secuencial	6
2.2	Paralelizado	7
2.3	Decisiones de diseño	9
3	Batería de pruebas	10
4	Evaluación del rendimiento: secuencial VS paralelizada	12
4.1	Variación del número de bloques	13
4.2	Variación del tamaño de bloque	14
4.3	Evaluación 1	15
4.4	Variación del número de bloques (con cómputo extra)	16
4.5	Variación del tamaño de bloque (con cómputo extra)	17
4.6	Evaluación 2	18
5	Compilación y ejecución del programa	19
6	Conclusiones.....	20

TABLAS DE PRUEBAS

Tabla 1: Prueba 1	10
Tabla 2: Prueba 2	11
Tabla 3: Prueba 3	11
Tabla 4: Tiempos 1	13
Tabla 5: Tiempos 2	14
Tabla 6: Tiempos 3	16
Tabla 7: Tiempos 4	17

ÍNDICE DE GRÁFICAS

Ilustración 1: Gráfica rendimiento 1	13
Ilustración 2: Gráfica rendimiento 2	14
Ilustración 3: Gráfica rendimiento 3	16
Ilustración 4: Gráfica rendimiento 4	17

1 Introducción

En este documento se procede a la explicación de la práctica 1, "*Concurrencia basada en tareas en C++11*" de la asignatura Desarrollo de Software de Sistemas del grado en Ingeniería Informática de la Universidad Carlos III de Madrid. La realización de la misma la han llevado a cabo los alumnos Javier Prieto Cepeda y Saúl Alonso Monsalve.

En primer lugar, se describirá el código, tanto en la versión secuencial como en la versión paralelizada, indicando las decisiones de diseño pertinentes para la correcta implementación de la práctica (incluyendo conceptos de programación genérica).

Una vez descrito el código, se realizará una batería de pruebas para comprobar el correcto funcionamiento de la práctica. Tras esto, se procederá a comentar un análisis de los resultados obtenidos y del rendimiento conseguido, explicando los motivos de los mismos y comparando las diferencias entre las versiones secuencial y paralelizada.

A continuación, se incluye una sección para indicar cómo compilar y ejecutar el programa entregado.

Por último, se realizarán una serie de valoraciones y conclusiones sobre la realización de la práctica.

2 Descripción del código

En este apartado se va a describir el código, tanto en la versión secuencial como en la versión paralelizada. En ambas versiones, se pretende implementar un sistema que contabilice el número de coincidencias en un vector de pares de elementos, entre un patrón dado por parámetro y los pares de elementos del vector.

Ambas versiones, tienen en común el comienzo del código. El código contiene un vector de pares de elementos que contendrá en cada posición un vector de pares de datos tipo float. Este vector, será rellenado con números aleatorios del 0 al 9 mediante la función *rand()* tantas veces como indique la constante **NUM_ELEMENTS** situada en el fichero **skeleton.hpp** de la documentación entregada. También existe una lista de dos elementos, con nombre *pattern*, que servirá como patrón para comprobar cuántas veces aparece dicho patrón en el vector.

Una vez rellenos los pares de elementos aleatorios del vector, se procede a calcular el número de *chunks* (bloques) en los que se dividirá el vector, los cuales se utilizarán para contabilizar el número de veces que el patrón aparece en el vector. El tamaño de cada bloque o chunk viene indicado en la constante **CHUNK_SIZE** situada en el fichero **skeleton.hpp** de la documentación entregada.

2.1 Secuencial

Tras calcular el número de chunks o bloques en los que se dividirá el vector, se procede a calcular de forma secuencial el número de coincidencias del patrón en cada uno de los chunks en que se divide el vector original de elementos. Para realizar ésta contabilización se llama a la función *match()*, a la cual se le introduce como argumentos el bloque seleccionado y la lista con el patrón.

Como resultado, esta función devolverá un vector de tipo *bool* del mismo tamaño que el vector chunk en el cual cada posición, corresponderá a "true" o "false" en función de si la coincidencia del patrón con la posición del vector chunk es verdadera o falsa, respectivamente. Una vez obtenido el vector de coincidencias, se llamará a la función *reduce*, para contar el número de posiciones con valor "true" del vector devuelto por la función anterior.

Esta función recibirá por parámetro el vector de coincidencias correspondiente a cada bloque (devuelto por la función *match*), y contabilizará el número de "trues" que posee dicho vector, devolviendo un entero con la suma. El valor devuelto por esta función será almacenado en un vector de enteros con nombre *partial_results* mediante la función *push_back()*.

Tras realizarse las funciones *match()* y *reduce()* para cada bloque, se realizará una llamada a la función *reduce2()*, la cual recibe por parámetro el vector de enteros que posee en cada posición el número de coincidencias de cada bloque, encargándose de sumar todos los valores almacenados en el vector, y devolviendo un entero, que será el número total de coincidencias en los pares de elementos del vector original con el patrón dado.

2.2 Paralelizado

El objetivo de esta sección es introducir mecanismos de paralelización y sincronización a nivel de hilos utilizando *packaged_task*. Para ello hemos contado, además, con un vector de futuros. En este vector de futuros se almacenarán los resultados obtenidos al contabilizar las coincidencias.

Tras calcular el número de bloques al igual que en la sección secuencial, se procede a crear cada una de las tareas a realizar (*packaged_task*), es decir, llamamos a una función auxiliar creada por nosotros llamada *spawn_task()*, la cual crea una tarea del tipo *packaged_task* indicando la función a ejecutar (*match()*) y se crea un hilo que ejecute la tarea. La función devuelve el futuro correspondiente a la tarea creada. Se creará una tarea por bloque de elementos.

Cabe destacar que en esta versión del código, la función *match()* ya no devuelve un vector de booleanos con las coincidencias, sino que antes de terminar llama a la función *reduce()* y devuelve el entero correspondiente al número de coincidencias que devuelve ésta. Hacemos esto para asegurarnos que el mismo hilo realiza tanto la función *match()* como la función *reduce()* del mismo bloque, una detrás de otra.

Ahora, la función *match()* será una función templatizada, que recibe un vector de vectores, que será el “chunk” correspondiente, y una lista que tendrá el patrón para la comprobación de las coincidencias. Los tipos de datos del patrón y del vector son genéricos, para que así esta función se pueda probar con distintos tipos de datos y no sólo con tipo float. La ventaja de utilizar plantillas es que los tipos los resuelve el propio compilador, no perdiendo eficiencia al ejecutar el programa. Además, esta función recorre tanto los pares de elementos del vector como el patrón con iteradores.

La función *reduce* en esta versión, se encuentra también templatizada, de forma que el tipo de los elementos del vector recibido es genérico para llevar a cabo la suma de coincidencias del vector. Esto produce una gran ventaja, ya que ahora podemos **juntar las funciones *reduce()* y *reduce2()* en una misma función**. Se encarga de acumular realizando un casting a entero los valores de cada elemento del vector recibido, devolviendo la suma total como un

valor de tipo entero. Así, esta función realiza la suma de todos los elementos del vector pasado por parámetros, pudiendo este vector ser un vector de enteros (antiguo `reduce2`) o un vector de booleanos (antiguo `reduce`).

Una vez la función *match* recibe el valor devuelto por la función *reduce()*, ésta dejará el futuro listo para ser recogido su valor en cualquier momento, siendo su valor el número de coincidencias totales en el chunk con el patrón establecido.

En la función principal (*main*), una vez lanzadas todas las tareas a realizar por los futuros, se realizará la petición de todos los valores correspondientes a los futuros almacenados el vector de futuros descrito anteriormente, de forma que estos valores se obtendrán una vez cada tarea haya sido debidamente completada. Una vez obtenidos los resultados de todos los futuros, es decir, las contabilizaciones de coincidencias de cada "chunk" que se almacenarán en un vector de tipo entero de tamaño igual al número de "chunks", se realizará desde la función principal una llamada a la función *reduce*, pasando como parámetro el vector con el número de coincidencias de cada "chunk". En éste caso, la función *reduce* devolverá el número total de coincidencias, realizando la suma de cada uno de los elementos del vector.

2.3 Decisiones de diseño

En cuanto a las decisiones de diseño del código contempladas, se pueden destacar las siguientes:

- **Uso de plantillas en la función *match()*** para poder comprobar las coincidencias con vectores de cualquier tipo de dato numérico (char, int, double, etc) y no solo de float.
- **Uso de iteradores en la función *match()*** para así poder introducir por parámetros vectores de cualquier tamaño y, además, tener patrones de diferentes longitudes (no sólo patrones de dos elementos).
- **Llamar a la función *reduce()* dentro de la función *match()*** para así asegurarse que el hilo que ejecute la tarea *packaged_task* correspondiente a la función *match()* ejecute también la función *reduce()*.
- **Uso de plantillas en la función *reduce()***, para así juntar las funciones *reduce()* y *reduce2()* en una única función, ya que las dos tenían la misma función pero variaban en el tipo de datos del vector de entrada.
- **Uso de iteradores en la función *reduce()*** para así poder sumar todas las posiciones de un vector de cualquier tamaño.
- Todos los argumentos de las funciones que realizan los hilos se pasan **por referencia** y no por valor, para así no tener que hacer copias y ganar rendimiento.
- Todos los **hilos** creados son **independientes** (detach) para así no tener que esperar por su terminación.
- **La secuencia de números generados con la función *rand()* es siempre la misma.** Si se quiere tener siempre una secuencia diferente se recomienda reiniciar la semilla con el valor del reloj de la máquina en la que se ejecute el programa.

3 Batería de pruebas

En este apartado, vamos a proceder a exponer las pruebas realizadas sobre el código para la obtención de los resultados de los apartados anteriores.

En este apartado comprobaremos que la versión concurrente consigue los mismos resultados que versión secuencial.

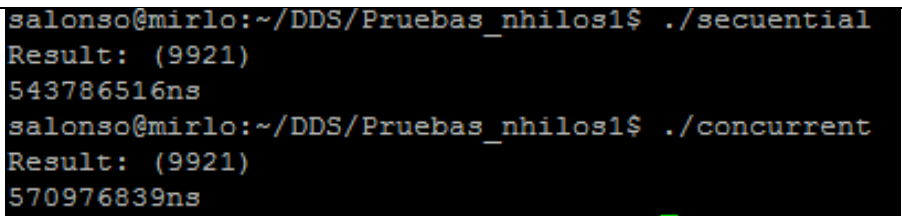
PRUEBA 1	
Objetivo	Comprobar que la versión concurrente produce el mismo resultado que la versión secuencial.
Entradas	NUM_ELEMENTS 1000000 CHUNK_SIZE 15625 Pattern1 3 Pattern2 1
Salida	Ambas versiones producen el resultado 9921.
Conclusión	Obtenemos el resultado esperado.
Captura pantalla	de 

Tabla 1: Prueba 1

PRUEBA 2	
Objetivo	Comprobar que la versión concurrente produce el mismo resultado que la versión secuencial.
Entradas	NUM_ELEMENTS 2000000 CHUNK_SIZE 1000000 Pattern1 7 Pattern2 5
Salida	Ambas versiones producen el resultado 20172.
Conclusión	Obtenemos el resultado esperado.
Captura pantalla de	<pre>salonso@mirlo:~/DDS/Pruebas_nhilos1\$./secuencial Result: (20172) 1097600230ns salonso@mirlo:~/DDS/Pruebas_nhilos1\$./concurrent Result: (20172) 1190935812ns</pre>

Tabla 2: Prueba 2

PRUEBA 3	
Objetivo	Comprobar que la versión concurrente produce el mismo resultado que la versión secuencial.
Entradas	NUM_ELEMENTS 8192 CHUNK_SIZE 1024 Pattern1 0 Pattern2 8
Salida	Ambas versiones producen el resultado 88.
Conclusión	Obtenemos el resultado esperado.
Captura pantalla de	<pre>salonso@mirlo:~/DDS/Pruebas_nhilos1\$./secuencial Result: (88) 4837222ns salonso@mirlo:~/DDS/Pruebas_nhilos1\$./concurrent Result: (88) 7264133ns</pre>

Tabla 3: Prueba 3

4 Evaluación del rendimiento: secuencial VS paralelizada

Nota: todos los tiempos medidos, tanto en la versión secuencial como en la versión concurrente, no han tenido en cuenta el tiempo de inicialización del vector de elementos, ya que consideramos que, como siempre se hace de forma secuencial, no aportaría resultados interesantes.

En este apartado se presentarán las gráficas y tablas de rendimiento, comparando tiempos en nanosegundos de diferentes pruebas en las versiones secuencial y concurrente.

Todas las pruebas se han realizado en una máquina con un procesador core i7 de 8 núcleos y con 36 GB de memoria RAM.

4.1 Variación del número de bloques

- **Número de elementos:** 1000000
- **Tamaño de bloque:** Desde 1000000 hasta 15625 elementos por bloque (para conseguir de 64 a 1 bloque, en potencias de dos).

Número de hilos	Tiempo secuencial (ns)	Tiempo concurrente (ns)
1 bloques	582555647	617877239
2 bloques	575881095	565489579
4 bloques	566105194	557998892
8 bloques	562744806	576317607
16 bloques	559012431	587160055
32 bloques	553936028	582761965
64 bloques	552642447	584339096

Tabla 4: Tiempos 1

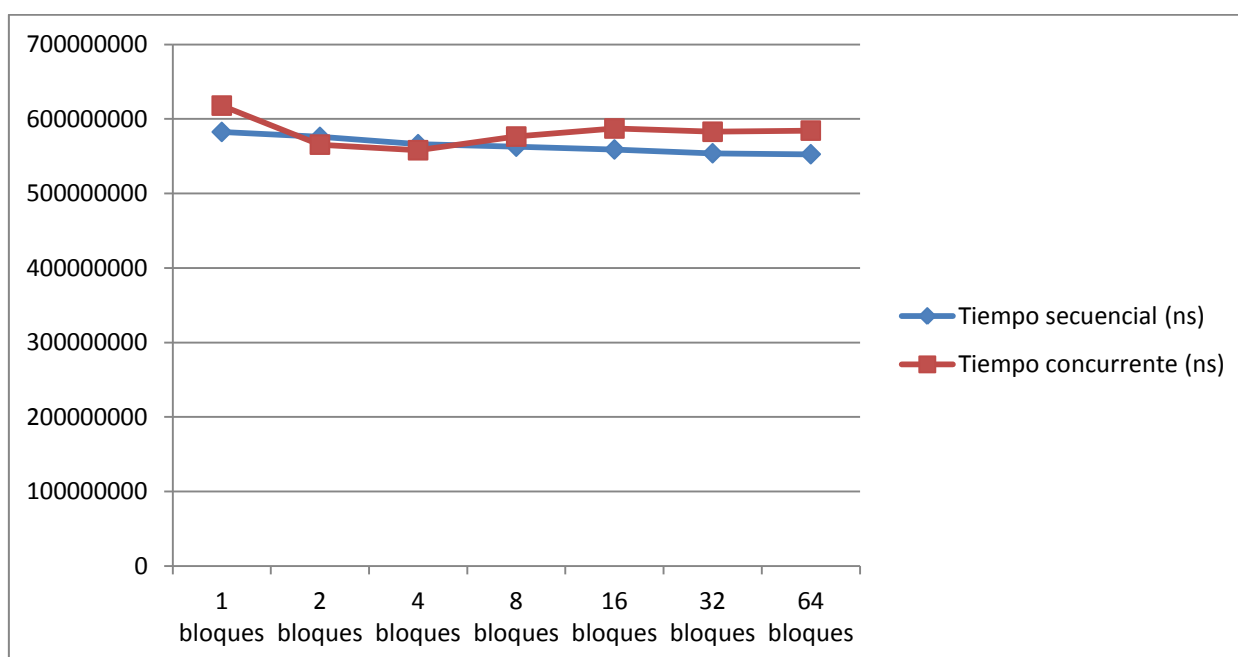


Ilustración 1: Gráfica rendimiento 1

4.2 Variación del tamaño de bloque

- **Número de bloques:** Siempre 8 bloques
- **Número de elementos:** Desde 1000000 a 64000000 elementos (para conseguir siempre 8 bloques).
- **Tamaño de bloque:** Desde 250000 hasta 16000000 elementos.

Elementos / bloque	Tiempo secuencial (ns)	Tiempo concurrente (ns)
250000 elementos	565881203	559307916
500000 elementos	1136735203	1121606416
1000000 elementos	2280807113	2241276607
2000000 elementos	4572612604	4373914705
4000000 elementos	9029135129	8845590863
8000000 elementos	18340722073	17597062780
16000000 elementos	36108133406	35269983845

Tabla 5: Tiempos 2

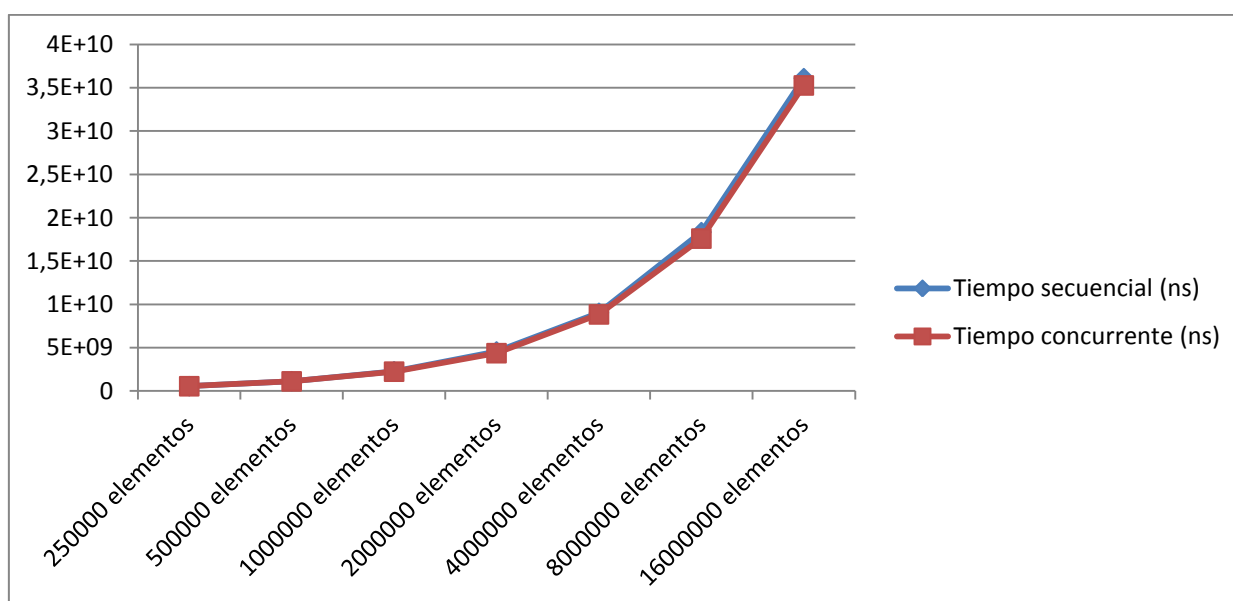


Ilustración 2: Gráfica rendimiento 2

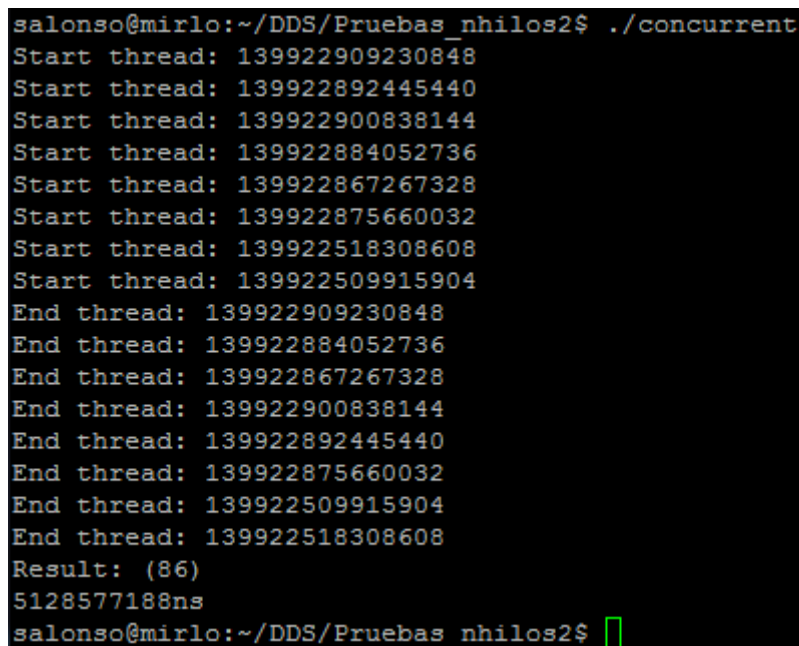
4.3 Evaluación 1

Como se puede ver en las dos gráficas y tablas anteriores, los tiempos en las versiones secuencial y concurrente son prácticamente idénticos. Tras analizar el código, hemos descubierto que esto se debe a que el proceso de creación de las tareas (crear un `packaged_task`, obtener el futuro, crear el hilo, etc) consume más tiempo que la propia función asociada a la tarea. Es decir, que el proceso de creación de las tareas y lanzar los hilos tarda más que las funciones `match()` y `reduce()`. Por lo tanto, para medir el rendimiento y comprobar que realmente la versión concurrente es más eficiente que la versión secuencial, hemos procedido a añadir el siguiente cómputo extra en la función `match()`, tanto en la versión secuencial como en la concurrente:

```
for(int i=0; i<200000000; ++i){
    borrar*=5;
    borrar/=5;
}
```

Ahora, hemos añadido cómputo suficiente como para que los diferentes hilos se ejecuten de forma concurrente.

Tal y como se puede comprobar en la siguiente captura de pantalla, ahora los hilos se ejecutan de forma concurrente, ya que todos empiezan antes de que alguno termine su ejecución:

A terminal window showing the execution of a program named 'concurrent'. The prompt is 'salonso@mirlo:~/DDS/Pruebas_nhilos2\$'. The program starts by printing 'Start thread:' followed by 10 unique thread IDs. Then it prints 'End thread:' followed by the same 10 thread IDs. The output shows that the threads start and end in a non-sequential order, indicating concurrent execution. Finally, it prints 'Result: (86)' and a timestamp '5128577188ns'. The prompt returns to 'salonso@mirlo:~/DDS/Pruebas_nhilos2\$' with a green cursor.

```
salonso@mirlo:~/DDS/Pruebas_nhilos2$ ./concurrent
Start thread: 139922909230848
Start thread: 139922892445440
Start thread: 139922900838144
Start thread: 139922884052736
Start thread: 139922867267328
Start thread: 139922875660032
Start thread: 139922518308608
Start thread: 139922509915904
End thread: 139922909230848
End thread: 139922884052736
End thread: 139922867267328
End thread: 139922900838144
End thread: 139922892445440
End thread: 139922875660032
End thread: 139922509915904
End thread: 139922518308608
Result: (86)
5128577188ns
salonso@mirlo:~/DDS/Pruebas_nhilos2$
```


4.4 Variación del número de bloques (con cómputo extra)

- **Número de elementos:** 1000000
- **Tamaño de bloque:** Desde 1000000 hasta 15625 elementos por bloque (para conseguir de 64 a 1 bloque, en potencias de dos).

Número de hilos	Tiempo secuencial (ns)	Tiempo concurrente (ns)
1 bloques	1962664795	1976433089
2 bloques	3345495062	1965066415
4 bloques	6015637922	2721525870
8 bloques	11469949568	5302188684
16 bloques	22380326212	10371209538
32 bloques	44364159218	20624757663
64 bloques	86872441481	41072750625

Tabla 6: Tiempos 3

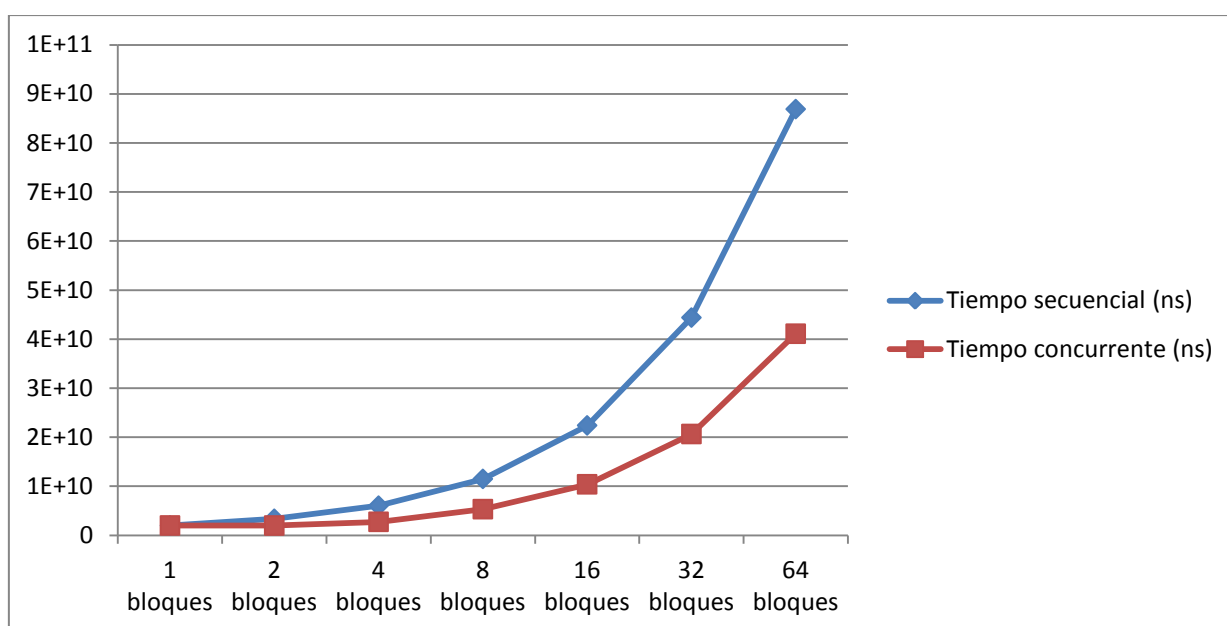


Ilustración 3: Gráfica rendimiento 3

4.5 Variación del tamaño de bloque (con cómputo extra)

- **Número de bloques:** Siempre 8 bloques
- **Número de elementos:** Desde 1000000 a 64000000 elementos (para conseguir siempre 8 bloques).
- **Tamaño de bloque:** Desde 250000 hasta 16000000 elementos.

Elementos / bloque	Tiempo secuencial (ns)	Tiempo concurrente (ns)
250000 elementos	6104770656	2723771300
500000 elementos	6675156958	3036580418
1000000 elementos	7809418547	3650667085
2000000 elementos	10085040138	5453891888
4000000 elementos	14392940947	9769133280
8000000 elementos	23329714586	19096882873
16000000 elementos	41194730903	35991635149

Tabla 7: Tiempos 4

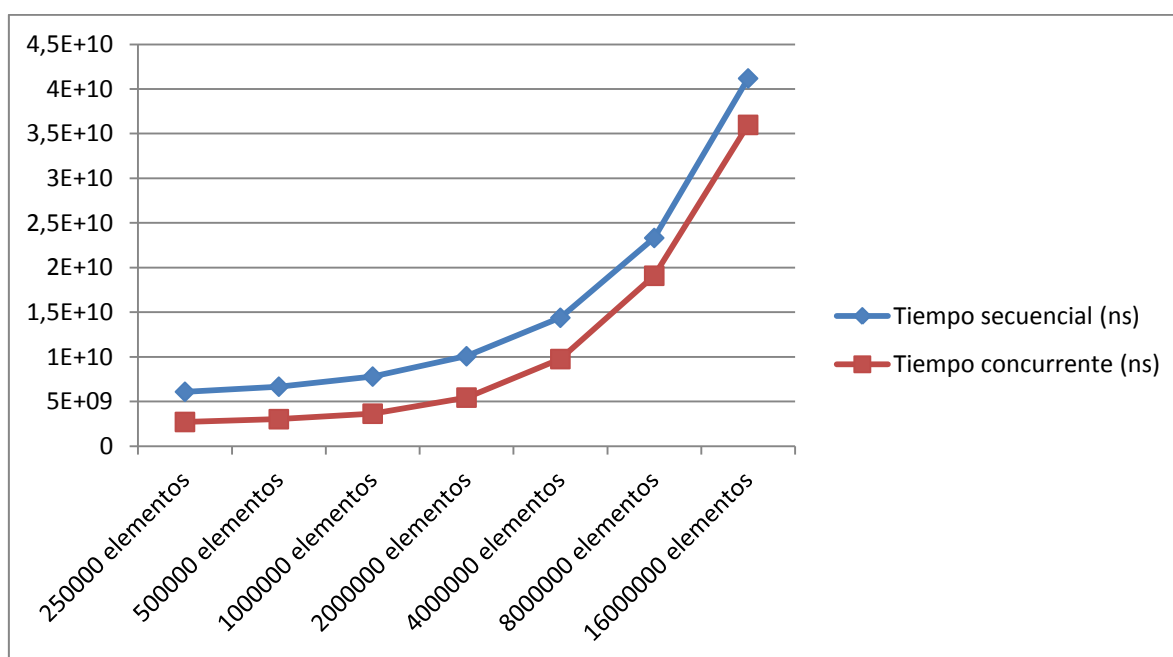


Ilustración 4: Gráfica rendimiento 4

4.6 Evaluación 2

Como se puede ver en las dos gráficas y tablas anteriores, ahora los tiempos de la versión concurrente mejoran los tiempos de la versión secuencial. Esto se debe a que ahora los hilos trabajan en paralelo en la versión concurrente, consiguiendo un speedup respecto a la versión secuencial.

Además, si el tamaño del vector fuera de varios órdenes de magnitud mayor y el tamaño de bloque permitiera que se creara un número de bloques no muy elevado (cercano a 8), las funciones *match()* y *reduce()* tendrían más tiempo de ejecución y, por lo tanto, comprobaríamos una aceleración aún mayor de la versión concurrente respecto a la secuencial.

5 Compilación y ejecución del programa

El código entregado cuenta con un fichero *Makefile*, con el cual se puede compilar el programa ejecutando el comando “make”, el cual creará el fichero ejecutable “skeleton”.

Para su ejecución, basta con teclear el comando “./skeleton”.

6 Conclusiones

Como conclusiones de esta práctica, cabe destacar la necesidad del conocimiento de los hilos en UNIX y los *packaged_task* de C++11. Esto se debe a que una vez realizado el análisis del rendimiento y comparar los tiempos de ejecución en la versión paralelizada y la versión secuencial, al observar que no se obtiene una mejora de tiempo al paralelizar el código, puede llevar a pensar en un posible error de implementación. Pero no se debe a ningún error, sino a que la parte del código paralelizada, conlleva a un tiempo de ejecución por hilo menor que el tiempo de creación de hilos, por lo que no se obtiene una mejora en tiempos totales.

También se debe destacar la importancia del conocimiento de la implementación de *templates* de C++11, lo cual resulta muy útil para la creación de funciones genéricas para que puedan funcionar con cualquier tipo de dato, evitando la sobrecarga de funciones con la complejidad que ello conlleva, obteniendo como resultado un código sencillo y elegante.

Uno de los problemas encontrados en la práctica, era la forma en que se realizaba el llenado del vector original, ya que si no se le define un tamaño fijo al principio, cuando el tamaño final de éste vector ha de ser de un tamaño considerable, el rendimiento obtenido es muy malo, ya que la inserción se realiza mediante la función *push_back()* de la clase vector, realizando una serie de realojamientos en memoria del vector causado por su variación de tamaño, lo cual causa una pérdida de tiempo considerable.

Por último, mencionar una posible mejora. Tras analizar el código se observa que la mayor parte del tiempo de ejecución del programa reside en la sección de inserción de elementos con valores aleatorios en el vector de elementos original, se podría obtener una mejora significativa de tiempos paralelizando esta sección del código, puesto que si este vector se dividiera en chunks al igual que en las otras secciones del código, se podría optimizar el llenado del vector.