PRÁCTICA 1

Memoria

Pilar Carrera Monterde

26 de Diciembre 2021

PREFACIO:

En esta práctica he conseguido todos los objetivos propuestos.
He recordado toda la programación de C, como también alguna parte de C++.

ÍNDICE:

1. **SISTEMA:**

Como he comentado en anteriores prácticas, mi computador personal cuenta con las siguientes características:

Todas las prácticas las he realizado sobre linux nativo en un portátil con 12 cores lógicos y 6 físicos.

- CPU AMD FX-8320E
- CPU 8 cores: 1 socket, 2 threads/core, 6 cores/socket.
- CPU @2.60GHz
- CPU: L2: 256K, L3: 12288K
- 16 GiB RAM
- Trabajo y benchmarks sobre SSD
- 172 procesos en promedio antes de ejecuciones
- Sin entorno gráfico durante ejecuciones
- ArchLinux 64bits actualizado

2. DISEÑO E IMPLEMENTACIÓN DEL SOFTWARE:

Esta primera práctica es en la que más hemos tenido que programar por nuestra cuenta, por lo que es dónde más decisiones de diseño he tenido que hacer.

Para poder unir todas las partes, he utilizado la compilación condicional, la cual, en un mismo código añadimos las tres funciones pedidas y algunas partes solo se realizan según la constante declarada previamente.

Al principio las realicé en distintos documentos, por lo que a continuación explicaré cada parte por independiente, aunque cuando lo entregue estará todo unido en un mismo documento.

1) Empezando por el base:

Decidí definir como variable global el acabar el programa satisfactoriamente o no, esto lo hice con un define, como muestro a continuación:

```
#define EXIT_FAILURE 1
#define EXIT_SUCCESS 0

double sumTot_g = 0;
double revTot_g = 0;

…
```

En este cuadro de código también muestro como puse globalmente las variables en las que iré sumando los elementos del array.

En el main, opté por calcular lo que tardaba la función con el gettimeofday al principio y al final de la función y, a continuación, restando el final menos el inicial.

```
struct timeval tiempoInicial;
gettimeofday(&tiempoInicial, NULL);

…

struct timeval tiempoFinal;
gettimeofday(&tiempoFinal, NULL);

time_t tiempoQueTardaSeg = tiempoFinal.tv_sec - tiempoInicial.tv_sec;
suseconds_t tiempoQueTardaMlSeg = tiempoFinal.tv_usec - tiempoInicial.tv_usec;
```

Lo primero que realicé en el main fue el control de parámetros, así comprobando que todo lo que nos han metido por consola está en orden: que el número de threads sea un int, que la función a realizar sea o suma o rev, que el número de elementos del array sea int, etc. Si se detecta algún parámetro mal introducido, se devuelve un mensaje de error y el programa termina con EXIT_FAILURE.

Tras comprobar que todo está en orden, creamos dinámicamente el array de elementos y reservamos espacio para el mismo con malloc, si no se ha podido reservar la memoria, mostramos un mensaje de error y terminamos el programa.

```
double* arrayElementos = (double*) malloc(numElementos * sizeof(double));

if (arrayElementos == NULL) {
        printf("No se ha podido reservar memoria correctamente");
        return(EXIT_FAILURE);
}
```

Tras reservar memoria, rellenamos el array como indica el enunciado con un for.
Analizamos el número de bloques que necesitaremos para repartir la suma de elementos entre ellos. Primero calculé el número de elementos por bloque necesarios, siendo estos el número de elementos entre el número de threads. Si esta división no es exacta, el resto de elementos debe ir al threads que antes termine. Esto lo he conseguido con esta condición:

```
float restoNumBloques = fmod(numElementos, numThreads);

…

if(restoNumBloques == 0)  {
        numElementosPorBloque = numElementos / numThreads;
 } else  {
        numElementosPorBloque = numElementos / numThreads;
        thread1MasTrabajo = 1;
  }
```

Gracias al join que haremos más adelante de los hilos, el primer hilo que entre en la función a realizar, será el primero que termine; por ello, esto lo controlaremos dentro de la dunción que realiza las operaciones, con una variable en la misma en la que, si es la primera vez que entra en la función, se ejecuta un cacho de código.

```
bool primeraVez = 1;

void funcionGeneral(...) {

        …

        if(primeraVez == 1) {
                primeraVez = 0;
                …
        }
}
```

El como saber que bloque debe ejecutar el hilo llamado, utilizo este algoritmo:

```
…

int posFinal = numElementosPorBloque;
int posInicial = 0;

int posInicial2 = numElementos - restoNumBloques;
int posFinal2 = numElementos;

for(int i = 0; i < numThreads; i++) {
        threads[i] = std::thread(funcionGeneral, funcionARealizar, arrayElementos,
                                                        posInicial, posFinal, posInicial2, posFinal2);
        posInicial = posFinal;
        posFinal = posFinal + numElementosPorBloque;
}

…
```

Como podemos ver, este for va creando el número de threads introducido por parámetro y llama a la función funcionGeneral indicándole que función debe realizar (sum o rev), pasándole el array de elementos creado anteriormente, la posición inicial de su bloque de elementos a sumar, la posición final y, finalmente, la posición inicial del bloque de elementos que sobra (el resto que comentamos anteriormente) y la posición final del mismo, siendo esta el número de elementos del array. Estos dos últimos parámetros solo se utilizarán una vez, en la primera vez que se llame a la función.

He optado por, en vez de hacer dos funciones, rev o sum, hacer una sola pero que, dependiendo qué valor le pases en el parámetro de funcionARealizar, hace una u otra.

```
…

if(strcmp(funcionARealizar, sumFunct) == 0) {
        std::lock_guard<std::mutex> guard(mutex_g);
        sumTot_g = sumTot_g + arrayElementos[i];
} else {
        std::lock_guard<std::mutex> guard(mutex_g);
        revTot_g = revTot_g - arrayElementos[i];
}

…
```

Hacemos el join de todos los hilos e imprimimos por pantalla el total calculado por la función y cuánto tiempo ha tardado.

2) *Logger:*

Partimos del código de base. Esta vez debemos añadir una función que nos sume paralelamente los hilos y avise al main cuando termine, si la suma del logger y del base son iguales, entonces está bien realizado.

Para hacer esto, he creado una estructura con en su interior un array que indica que hilos han sido usados y cuales no. Esto lo he hecho para ir apuntando el resultado de los hilos individualmente para después poder sumarlos en el logger. Esta estructura la inicializo a 0,5 y el logger comprobará que hilos han escrito (porque no siempre van a ser los 12) comparando si el valor de la posición del array es 0,5 u otro valor distinto.

Aquí creo la estructura:

```
…

typedef struct estructHilos {
        double valorHilosUsados[12];
} estructHilos;

…
```

En la funcionARealizar, cada vez que se calcula un dato, se añade a la posición del hilo que acaba de escribir:

```
…

if(strcmp(funcionARealizar, sumFunct) == 0) {
        estructHilos->valorHilosUsados[numHilo] = sumaParcialHilo;
  } else {
        estructHilos->valorHilosUsados[numHilo] = revParcialHilo;
}

…
```

Para poder realizar esto, tuve que añadir un nuevo parámetro a funcionARealizar, siendo este el número de hilo que llamaba a la función. SumaParcialHilo es una variable local de funcionARealizar en la cual se guarda el valor calculado en ese hilo.

Para poder hacer que el main tenga que esperar a que el logger termine de sumar y para que el funcionARealizar despierte al logger cada vez que se le llame, tuve que utilizar las variables condicionales y el mutex.

Solo he utilizado un mutex y una variable condicional:

```
…

std::mutex mutex_g;
std::condition_variable CVLogger;

…
```

Y, para poder despertar al logger y al main utilicé dos booleanos:

```
…

bool despiertaLoger_g = 0;
bool terminaSuma = 0;

…
```

Siendo el primero el que despierta al logger y el segundo el que avisa al main cuando el logger termina de sumar.

El logger es una función a la cual se le pasan los argumentos de la estructura de los hilos, comentada anteriormente y la condición variable.

```
int logger(estructHilos* estructHilos, std::condition_variable* CVLoggerMain);
```

En su interior, nos encontramos un while, el cual se termina cuando el número del thread que llega al logger es igual al número de threads que nos han metido por parámetro. Dentro de este while, el logger se queda esperando por el wait a que le despierten desde funcionARealizar con la variable condicional comentada anteriormente: despiertaLoger_g.

```
...

while(numThreads != hiloLLegado) {

        std::unique_lock<std::mutex> ulk(mutex_g);
        CVLogger.wait(ulk, []{return despiertaLoger_g;});

        ...
}
```

Desde funcionARealizar se le despierta con el siguiente código:

```
  ...

  despiertaLoger_g = 1;
  CVLogger.notify_one();

...
```

En este fragmento de código cambiamos el booleano de 0 a 1 y hacemos notify a CVLogger, la misma variable condicional que tiene en el logger el .wait.

Así, el logger se despierta y vuelve a comprobar la variable despiertaLoger_g.

Al salir del while, recorremos el array donde hemos ido acumulando el resultado de cada hilo independiente y, como se pide en el enunciado, se imprime por pantalla y se escribe en un fichero llamado dumb.log. Tras esto se suman todos los resultados parciales y se avisa al main que el logger ha terminado de sumar:

```
...

terminaSuma = 1;
CVLoggerMain->notify_one();

...
```

En el main, hemos tenido que añadir la creación de logger, de la segunda variable condicional pasada como parámetro y su join:

```
…

std::condition_variable CVLoggerMain;

std::thread loggerHilo(logger, &estructuraHilos, &CVLoggerMain);

…

loggerHilo.join();

…
```

El main debe esperar a que le despierte el logger al terminar, y esto se hace igual que antes, con la variable condicional y el mutex:

```
…

std::unique_lock<std::mutex> ulk(mutex_g);
CVLoggerMain.wait(ulk, []{return terminaSuma;});

…
```

Al despertar al main, se comprueba si lo calculado por el base y por el logger es igual, si no es igual se da un mensaje de error, si es correcto, se imprime lo que vale el hilo resultante y el programa termina.

3) *Optimización:*

Parte del código del logger y el base.

Para realizar esto, debido a que puse el mutex en funcionARealizar dentro del for, retrasaba el tiempo de ejecución considerablemente, por lo que lo saqué del for y añadí dos variable atómica:

```
…

std::atomic<double> totSumaAt_g;
std::atomic<double> totRevAt_g;

…
```

Primero compruebo si está libre, y después realizo la operación sobre la variable global.

```
…

totSumaAt_g.is_lock_free();
totSumaAt_g = totSumaAt_g + arrayElementos[i];

…
```

El cómo afecta al tiempo de ejecución lo hablaré en el siguiente apartado.

4) *Conector con java:*

Para el conector de java tenemos que cambiar 3 archivos, P1.java, P1Bridge.cpp. y P1Bridge.java.

En el P1.java debemos tratar los argumentos de entrada como antes lo hacíamos en el cpp pero en java. Tras esto, si todo ha ido correctamente, debemos crear un nuevo objeto de la clase P1Bridge y meterle los argumentos tratados:

```
…

(new P1Bridge()).compute(numElementos,funcionARealizar,numThreads);

…
```

Para saber si todo ha ido bien, he creado un booleano llamado todoBien, si este es 1, significa que algo ha salido mal e imprime por pantalla un error; en cambio, si todoBien es 0, crea el nuevo objeto de la clase, como hemos mostrado anteriormente.

En la clase P1Bridge.java hacemos que, todo lo creado anteriormente, se introduzca en el cpp como una librería llamada p1bridge y llama a la función compute del P1Bridge con los argumentos guardados anteriormente.

```
…

public native void compute(int numElementos, String funcionARealizar,int numThreads);

…
```

Dentro del P1Bridge.cpp, tenemos la función proporcionada Java_P1Bridge_compute, en la cual añadimos los siguientes argumentos:

```
JNIEXPORT void JNICALL Java_P1Bridge_compute(JNIEnv *env, jobject thisObj,
                        jint numElementos, jstring funcionARealizar, jint numThreads) {

…
```

Siendo estos jint numElementos, jstring funcionARealizar y jint numThreads. Estos argumentos se introducen como parámetros de java y debemos pasarlos a parámetros de C++, esto solo nos hace falta hacerlo con el jstring, ya que en c debemos unas char.

La transformación lo hacemos con:

```
...

const char *funcionARealizar2 = env->GetStringUTFChars(funcionARealizar, 0);
char *funcionARealizar3 = strdup(funcionARealizar2);

...
```

En estos pasos, cogemos la cadena nativa de java y la transformamos a un puntero a char.

He creado una función llamada todo con parámetros numElementos, funcionARealizar y numThreads, en la cual he copiado y pegado el código de base creado en la parte 1.

Después, liberamos el espacio ocupado por la cadena usada.

```
...

env->ReleaseStringUTFChars(funcionARealizar, funcionARealizar2);

...
```

3. METODOLOGÍA Y DESARROLLO DE LAS PRUEBAS REALIZADAS

a) Base:

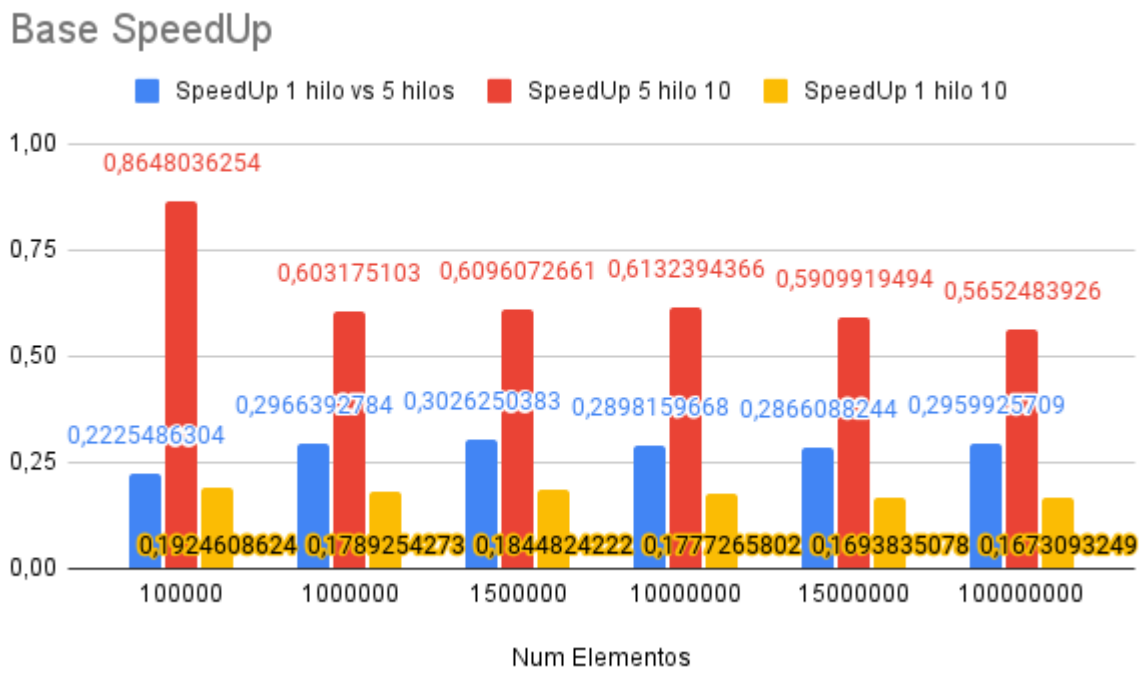
Como veremos, los speedup de 1 hilo frente a 5 o 10, son malos. Esto se debe a lo comentado anteriormente, el mutex que he puesto dentro del bucle for en vez de fuera. Por ello, en este apartado no mejora mucho el tiempo de ejecución, incluso se retrasa, y por ello los resultados en los speedUps son muy bajos.

Los tiempos serían estos:

num hilos	numElementos	Tiempo ejecucion
1	100000	0,002803
1	1000000	0,028413
1	1500000	0,044442
1	10000000	0,285811
1	15000000	0,419722
1	100000000	2,862910

5	100000	0,012595
5	1000000	0,095783
5	1500000	0,146855
5	10000000	0,986181
5	15000000	1,464442
5	100000000	9,672236
10	100000	0,014564
10	1000000	0,158798
10	1500000	0,240901
10	10000000	1,608150
10	15000000	2,477939
10	100000000	17,111479

Y las gráficas de speedUp las siguientes:



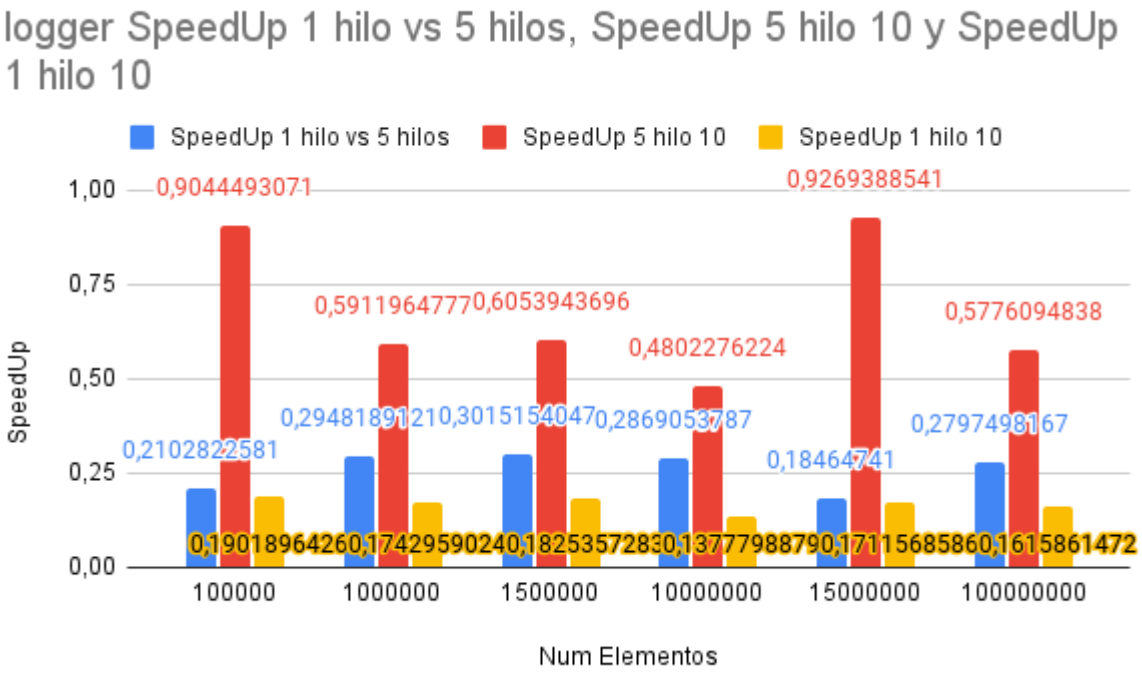
Como podemos ver, nunca supera el 1, por lo que es incluso peor que con solo un hilo, esto cambiará en la parte de optimización.

b) Logger:

Para el logger calcularé el tiempo que tarda y el speedUp respecto a los tiempos anteriores, volvemos a observar que el speedUp es nulo, por el mismo fallo que comenté en el base, siendo la tabla de tiempos:

num hilos	numElementos	Tiempo ejecucion
1	100000	0,003129
1	1000000	0,031234
1	1500000	0,048727
1	10000000	0,307323
1	15000000	0,454423
1	100000000	3,085683
5	100000	0,014880
5	1000000	0,105943
5	1500000	0,161607
5	10000000	1,071165
5	15000000	2,461031
5	100000000	11,030152
10	100000	0,016452
10	1000000	0,179201
10	1500000	0,266945
10	10000000	2,230536
10	15000000	2,655009
10	100000000	19,096210

Siendo los gráficos de tiempos:



Volvemos a ver que los tiempos son malos, y si hacemos el speedUp de base respecto al logger, no hay gran mejora y los tiempos siguen muy parecidos

num hilos	numElementos	SpeedUp Base vs Logger
1	100000	0,895813
1	1000000	0,909682
1	1500000	0,912061
1	10000000	0,930002
1	15000000	0,923637
1	100000000	0,927804
5	100000	0,846438
5	1000000	0,904099
5	1500000	0,908717
5	10000000	0,920662
5	15000000	0,595052
5	100000000	0,876891
10	100000	0,885242
10	1000000	0,886145
10	1500000	0,902437
10	10000000	0,720970

10	15000000	0,933307
10	100000000	0,896067

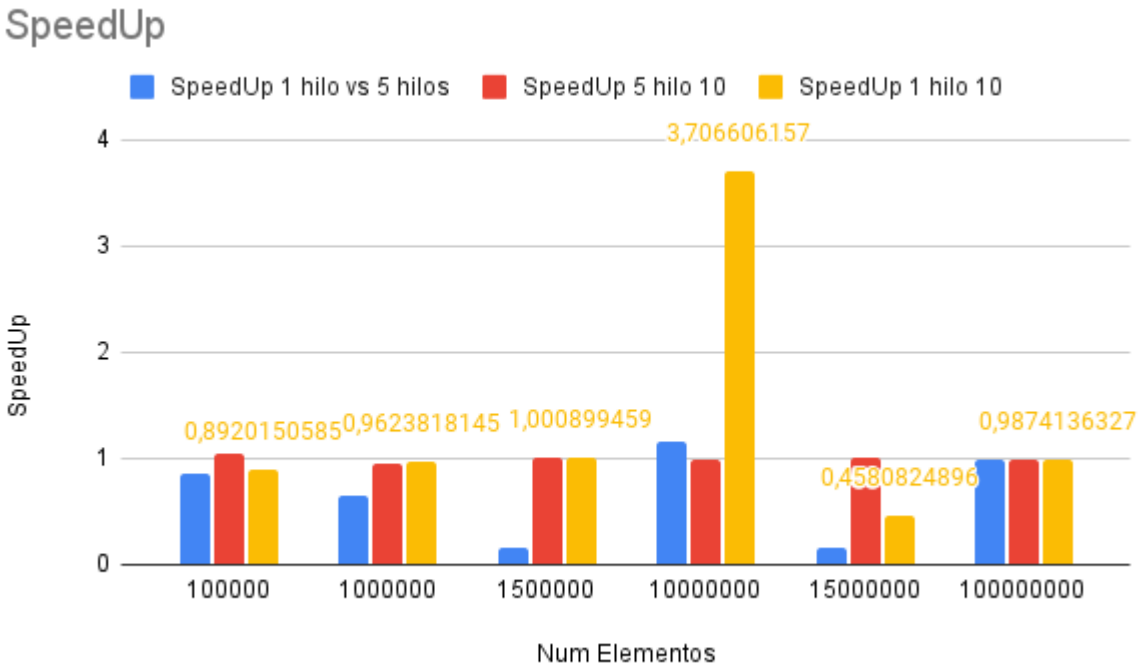
c) Optimización:

En este apartado es cuando veremos significativamente resultados, ya que para poder optimizarlo saqué el mutex de dentro del bucle. Voy a hacer una comparativa de speedUp entre 1 hilo, 5 y 10, igual que antes, y una comparativa con el base y el logger respecto a tiempos.

El tiempo que tarda por cada elemento e hilo en una tabla es:

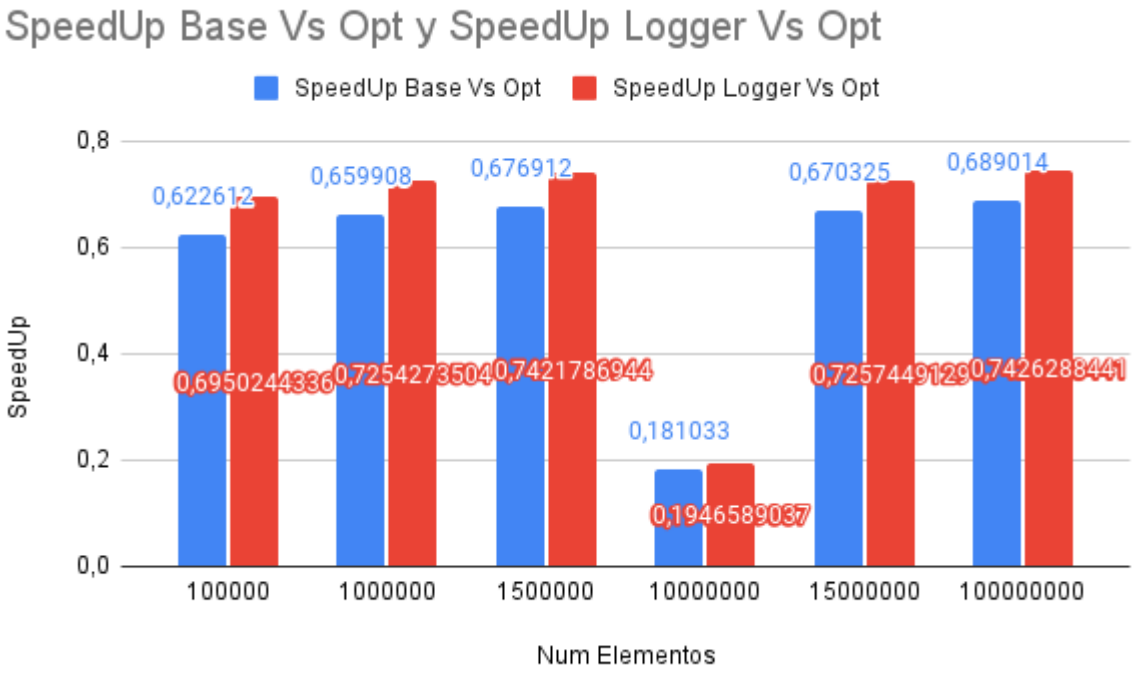
num hilos	numElementos	Tiempo ejecución
1	100000	0,004502
1	1000000	0,043056
1	1500000	0,065654
1	10000000	1,578777
1	15000000	0,626147
1	100000000	4,155081
5	100000	0,005286
5	1000000	0,042408
5	1500000	0,065689
5	10000000	0,422322
5	15000000	1,366191
5	100000000	4,172228
10	100000	0,005047
10	1000000	0,044739
10	1500000	0,065595
10	10000000	0,425936
10	15000000	1,366887
10	100000000	4,208045

Y, en speedUp respecto a cada hilo, como hemos hecho anteriormente, tenemos:

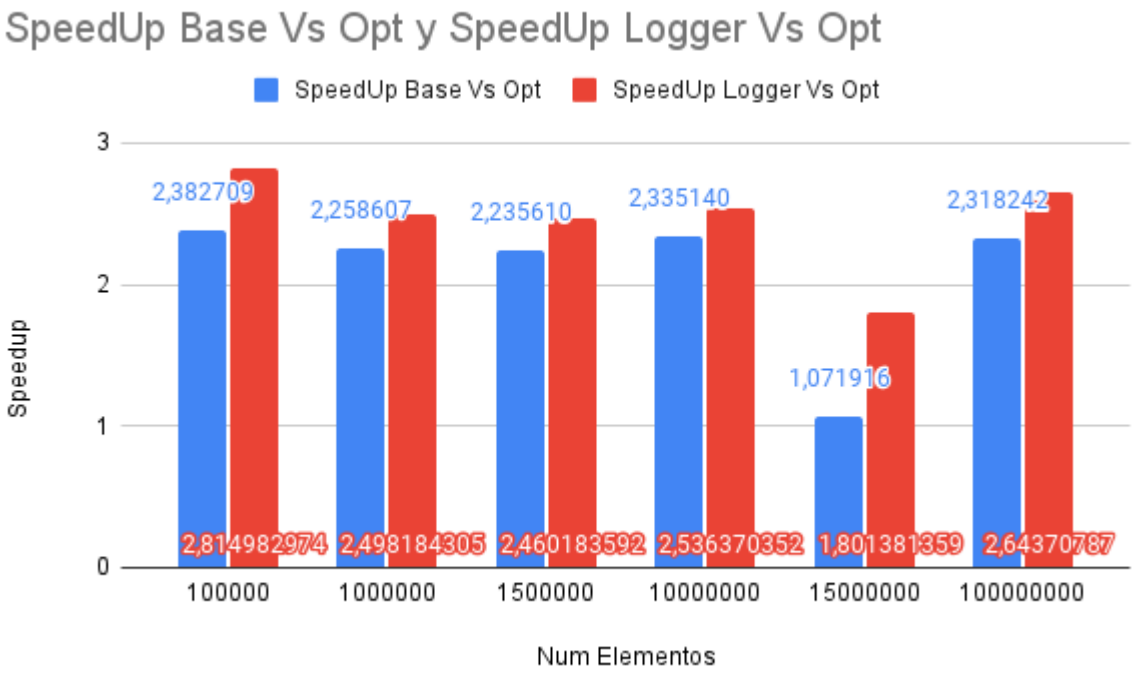


Podemos observar una mejora significativa con el número de elementos 10.000.000, respecto a 1 hilo contra 10, con un speedUp de hasta 3,7. En general vemos mayor speedUp y cambios que anteriormente, esto se debe al mutex que saqué de dentro del bucle.

Si lo vemos respecto a Base y Logger, vemos que con un hilo el speedUp no es muy diferente, ya que es lo mismo con mutex dentro o fuera, no afecta.

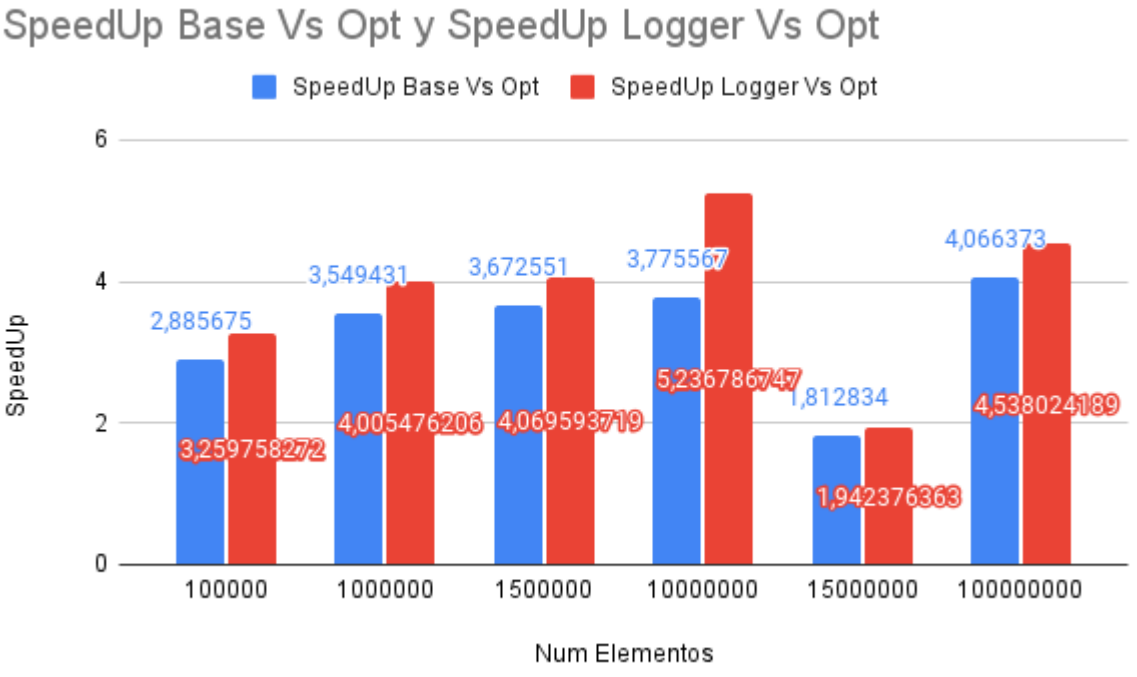


Si lo comparamos con 5 hilos ya empezamos a ver una mejora significativa:



Con un speedUp de hasta 2,8. Esto demuestra el buen trabajo que se ha hecho al optimizar el código. Al sacar el mutex de dentro del bucle, no produce un cuello de botella y el código va mucho más fluido.

Si vemos la comparativa con 10 hilos, aumenta aún más:



d) Conector con java:

Debemos analizar el tiempo que tarda en ejecutarse el base en java respecto a c++.

Este speedUp no es muy favorecedor, por el mutesx que comentamos antes. Para poder hacer bien las pruebas cambié el mutex fuera del bucle e hice el speedUp respecto a estos datos.

Vemos como el tiempo de ejecución del base mejora considerablemente

num hilos	numElementos	Tiempo ejecucion Base nuevo	Tiempo ejecucion Java nuevo	SpeedUp
1	100000	0,00082	0,0008890	0,9223847019
1	1000000	0,007134	0,0073230	0,9741909054
1	1500000	0,0125830	0,012706	0,9903195341
1	10000000	0,0757430	0,07433	1,019009821
1	15000000	0,1109530	0,112828	0,9833817847
1	100000000	0,7213160	0,735694	0,9804565485
5	100000	0,0009990	0,001103	0,9057116954
5	1000000	0,0087280	0,007569	1,153124587
5	1500000	0,0113090	0,011485	0,9846756639
5	10000000	0,0745900	0,073538	1,014305529
5	15000000	0,1167980	0,109655	1,065140668
5	100000000	0,7255480	0,723724	1,002520298
10	100000	0,0011990	0,001151	1,041702867
10	1000000	0,0081180	0,007629	1,064097523
10	1500000	0,0123090	0,011666	1,055117435
10	10000000	0,0765900	0,07269	1,053652497
10	15000000	0,1118510	0,1112	1,005854317
10	100000000	0,7338890	0,719895	1,019438946

Vemos un speedUp de hasta 1, esto implica que el base ejecutado por el conector tarda menos que el base en c++. Esto se debe a que los argumentos los administramos en el java y el tiempo de ejecución de ello se resta al tiempo total. Quitando esto, tardan parecidos.

Para calcular todos los tiempos he usado el gettimeofday, como indiqué al principio.

4. DISCUSIÓN:

Esta práctica, a mi punto de vista, ha sido la más entretenida de las 4, pero también la más costosa. Me encontré pensando en cómo resolver los problemas durante horas sin darme cuenta y, cuando lo resolvía, me ponía a programar instantáneamente.

Pero es una práctica muy larga, costosa y debería separarse en dos prácticas mínimo. Me llevó un mes completarla por completo.