Informe práctica 1

Zdravko Dimitrov Arnaudov
04/01/2021

Prefacio

La práctica en cuestión sirve como base para estructurar nuestros conocimientos acerca de la programación paralela. Además, nos ayudará a comprender y desarrollar con soltura las siguientes prácticas, enfocadas a la paralelización con OpenMP.
El objetivo a conseguir en esta actividad es alcanzar la paralelización mediante el paradigma de memoria compartida, usando primitivas y APIs de C++ y conocer tanto el potencial como los mecanismos de multithreading basado en C++/Pthreads.
Desde mi punto de vista, esta práctica ha supuesto un antes y un después en cuanto a la forma de abordar la programación. Se han destacado la importancia de varios conceptos como la optimización del código para no consumir demasiada memoria, una buena estructura para respetar la legibilidad y corrección, una depuración precisa y frecuente para revisar la funcionalidad, aprender a usar nuevas primitivas y APIs de C++ así como los mecanismos de programación paralela que establece Pthreads y adoptar nociones sobre benchmarking para contrastar resultados.
En mi opinión, la parte más complicada fue adentrarme a programar y leer código, a pesar de tener ideas claras sobre lo que debía hacer mi programa. Creo que no estábamos acostumbrados al nivel que requería la práctica pero es muy necesario. Al final, para poder aprender es imprescindible pelearse con el código, el tiempo que sea necesario. En cambio, la parte que me ha parecido más sencilla fue contrastar información y darle significado, según el comportamiento que ha adquirido mi computador a lo largo de la práctica.
Por último, aunque esta práctica haya sido muy extensa, no la habría modificado en absoluto. Quizás con más tiempo, habría realizado las partes optativas con tranquilidad pero aquello que me ha dado tiempo a completar, ha sido desde luego muy interesante y beneficioso.

Índice

1. Sistema

Con intención de obtener una descripción del sistema lo más detallada posible, buscaremos una serie de comandos equivalentes a '*cpuinfo*' o '*lscpu*', puesto que disponemos de Mac OS y dichas instrucciones no son reconocidas.

Uno de ellos se corresponde con '*system_profiler SPHardwareDataType'*, el cual nos muestra los aspectos hardware más generales y relevantes a tener en cuenta:

```
Hardware Overview:

      Model Name: MacBook Pro
      Model Identifier: MacBookPro17,1
      Processor Speed: 2,4 GHz
      Number of Processors: 1
      Total Number of Cores: 8
      L2 Cache (per Core): 4 MB
       Memory: 8 GB
```

Otro de los comandos encontrados es '*sysctl -a* | *grep machdep.cpu* ' y este en particular nos aporta información adicional sobre el número de cores físicos y lógicos del procesador:

```
machdep.cpu.cores_per_package: 8
machdep.cpu.core_count: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
machdep.cpu.brand_string: Apple M1
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR PGE MCA
CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS HTT TM PBE SSE3 PCLMULQDQ
DTSE64 MON DSCPL VMX EST TM2 SSSE3 CX16 TPR PDCM SSE4.1 SSE4.2 AES SEGLIM64
machdep.cpu.feature_bits: 151121000215084031
machdep.cpu.family: 6
```

Por último, emplearemos un tercer comando '*sysctl hw*' para observar más en detalle el reparto de cores y también información adicional asociada a la memoria cache:

```
hw.ncpu: 8
hw.byteorder: 1234
hw.memsize: 8589934592
hw.activecpu: 8
hw.physicalcpu: 8
hw.physicalcpu_max: 8
hw.logicalcpu: 8
hw.logicalcpu_max: 8
hw.cputype: 7
hw.cpusubtype: 4
hw.cpu64bit_capable: 1
hw.cpufamily: 1463508716
hw.cpusubfamily: 0
hw.cacheconfig: 8 1 1 0 0 0 0 0 0 0
hw.cachesize: 3612786688 65536 4194304 0 0 0 0 0 0 0
hw.pagesize: 4096
hw.pagesize32: 16384
hw.busfrequency: 100000000
hw.busfrequency_min: 100000000
hw.busfrequency_max: 100000000
hw.cpufrequency: 2400000000
hw.cpufrequency_min: 2400000000
```

```
hw.cpufrequency_max: 2400000000
hw.cachelinesize: 64
hw.l1icachesize: 131072
hw.l1dcachesize: 65536
hw.l2cachesize: 4194304
hw.tbfrequency: 24000000
```

2. Diseño e implementación del Software

A continuación, se explicarán las pautas que hemos ido tomando para diseñar la estructura del programa. Antes de ponernos a programar, lo primero que hicimos fue hacer un pseudo-código para dividir nuestro trabajo en secciones y determinar qué completar por orden de prioridad. De esta forma, el orden que se siguió para avanzar con la práctica fue: estructura de datos, obtención de los argumentos, inicialización, ejecución secuencial, ejecución multithread, logger y optimización

2.1. Estructura de datos

En relación a la estructura de datos, se ha interpretado que lo ideal es organizar la información en estructuras. Una de ellas servirá para recoger los argumentos de entrada del programa, que llamaremos 't_args'.

```
typedef struct args {
        long N;
        t_operation op;
        t_mode m;
        int n_threads;

} t_args;
```

Como podemos observar, sus campos almacenan la longitud del array de elementos, el tipo de operación (suma, resta o xor), el modo de ejecución (single o multithread) y el número de threads en caso de seleccionar multithreading. Para los campos de operación y modo, tenemos enumerados para hacer el código más legible y óptimo.

```
typedef enum operation {SUM=0, SUB, XOR} t_operation;
typedef enum mode {single, multi} t_mode;
```

Nuestra segunda estructura del programa 't_data' será la que contemple toda la gestión para el procesado del array de elementos. A medida que progresamos con el código, esta se irá expandiendo con más campos según avanzamos con multithreading.

```
typedef struct data{
        double *array_datos;
         long size;
        double resultado;

} t_data;
```

Para poder acceder al array en cuestión, uno de los campos será un puntero al espacio de memoria que reservaremos dinámicamente con malloc, cuyo proceso tendrá lugar en la inicialización una vez hayamos recogido los argumentos correctamente. También disponemos de un campo 'resultado' que almacenará tanto el cómputo secuencial como paralelo de los threads. Los atributos que aparecerán en la versión final de la estructura se detallarán a medida que avanzamos con las explicaciones.

```
typedef struct data{
        double *array_datos;
         long size;
        double resultado; //resultado globlal

        /*En caso de usar optimize*/
        std::atomic<double> resultado_atm;

        //Para delimitar área de trabajo de thread
        long istart;
        long elementos_thread;

        /*en caso haber exceso*/
        bool exceso_gestionado;
        int tam_exceso;

        /*en caso de emplearse logger*/
        std::thread logger;
        bool thread_ready;
        double *array_logger;
        double resultado_parcial;
        double resultado_logger;
        int identificador_thread;

        /*comunicación entre main y logger*/
        std::condition_variable cv_logger_main;
        bool logger_ready;
        bool main_ready;

}t_data;
```

2.2. Obtención argumentos

Después de haber abordado la estructura de datos, comenzamos a recoger los argumentos del programa uno a uno y asegurándonos de que son correctos. Al ver que los datos se almacenaban correctamente, mostrándolos por pantalla, decidimos implementar una gestión de errores contundente para evitar que el usuario cometiese fallos comunes.

```
        ...
//mode
if ((strcmp(argv[3], "single"))==0){
        argumentos.m=single;

} else if ((strcmp(argv[3], "multi"))==0){
        argumentos.m = multi;
```

```
} else {
        printf ("El modo de operación no es válido (únicamente single o multi).\n");
        return -1;

}
        ...
```

2.3. Inicialización

Al habernos asegurado de que los argumentos están correctamente introducidos y son válidos, podemos proceder con la inicialización de la estructura 't_data'.
Para cumplir con esta tarea, disponemos de una función que inicializa cada campo.

```
void inicializa_datos(t_data *datos, t_args *argumentos){

        datos->array_datos = (double*) malloc(argumentos->N*sizeof(double));

        for (int i = 0; i < argumentos->N; ++i){
                datos->array_datos[i] = i;
        }

        #ifdef FEATURE_OPTIMIZE
        datos->resultado_atm  = 0.00;
        #endif

                datos->size = argumentos->N;
        datos->exceso_gestionado = false;
        datos->tam_exceso = 0;
        datos->resultado = 0.00;
        datos->istart = 0.00;
        datos->elementos_thread = 0.00;

}
```

A simple vista, podemos ver como se reserva dinámicamente el espacio justo necesario para albergar el array de elementos según sus dimensiones. Seguidamente después, inicializamos cada elemento con el valor de su respectiva posición. Mas adelante, se reflejan otros campos inicializados que emplearemos en multithreading. También, se comienzan a apreciar instancias de compilación condicional, pero las abordaremos en su momento.

2.4. Single-thread

Cuando las estructuras están listas, podemos comenzar a computar. En este instante, hablaremos sobre ejecución condicional porque dependiendo del modo de ejecución que haya seleccionado el usuario, se computará de una forma u otra.

Nuestro primer objetivo será la ejecución single thread. Lo primero que debemos comprobar es qué tipo de operación se ha escogido para saber qué es lo que se debe

calcular. Después de este paso, toman lugar tres actividades. La primera de ellas, se corresponde con la operación en cuestión que debe computarse. La segunda, será la medición del tiempo de ejecución en nuestra región de interés (ROI), el cual se corresponde con el inicio y final del bucle. Finalmente, se imprimen los resultados por pantalla una vez hemos calculado la operación y anotado el tiempo.

```
if (argumentos.op==SUM) { //SUM

        gettimeofday(&start,NULL); //medimos tiempo inicial
        for (int i = 0; i < argumentos.N; ++i){
                datos.resultado += datos.array_datos[i];

        }
        gettimeofday(&end, NULL); //medimos tiempo final

        std::cout << "El resultado de la operación suma es: " << datos.resultado
<<std::endl;
        printf("El tiempo empleado en la operación son: %ld microsegundos\n",
((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec)));
```

Para poder recoger los tiempos de ejecución hemos empleado la API ‘gettimeofday()’, ubicada en <sys.time.h>, tal y como indicaba el enunciado. Aparte de la función, hemos instanciado la siguiente estructura con un campo de inicio y final. Por último, solo nos faltaba imprimir la diferencia del tiempos.

```
struct timeval start, end;
```

Al finalizar con la operación de suma, se ha repetido el mismo procedimiento para la resta y xor, solo cambiando la forma de computar.

2.5. Multi-thread

El siguiente objetivo en la lista es conseguir ejecución multithread. Lo primero que debemos hacer es realizar una serie de tareas previas. Primero, definimos un array de threads, tantos como haya especificado el usuario, mediante la API de C++ correspondiente recogida en <threads >. Después, obtenemos el número de elementos que debe operar cada thread en el array. Si el reparto es exacto, significa que no habrá ningún exceso por calcular y por tanto, mediante un booleano lo indicamos. Si no fuese así, la variable indicaría lo contrario y el exceso se calcularía por el thread más rápido en terminar.

```
std::thread threads[argumentos.n_threads];
datos.elementos_thread = (long) argumentos.N / argumentos.n_threads;

datos.tam_exceso = datos.size - (datos.elementos_thread*argumentos.n_threads);
if (datos.tam_exceso == 0){
        datos.exceso_gestionado = true;

}
```

A continuación, procedemos con la medición de la región de interés y la impresión de los resultados obtenidos. La nueva ROI difiere por completo de la anterior, ahora se tendrán que crear cada uno de los threads indicados, asignarles la región de trabajo correspondiente y esperar a que terminen. Para poder indicar la sección en la que trabaja cada thread, multiplicamos el valor del iterador i que recorre el bucle por el número de elementos que debe computar cada hilo. De este modo, nos aseguramos de que cada uno opere en su respectiva área de trabajo.

```
for (int i=0; i<argumentos.n_threads; ++i){

        if (argumentos.op == SUM){
                threads[i] = std::thread(procesa_SUM, i * datos.elementos_thread, i, &datos);

        } else if (argumentos.op == SUB){
                threads[i] = std::thread(procesa_SUB, i * datos.elementos_thread, i, &datos);

        } else{
                threads[i] = std::thread(procesa_XOR, i * datos.elementos_thread, i, &datos);

        }

}
```

Una vez creados los threads, esperaremos a que terminen de computar uno a uno.

```
        for (int i = 0; i < argumentos.n_threads; ++i){
                threads[i].join();

        }
```

Cuando los threads hayan acabado, el main accederá al campo ‘resultado’ para imprimir por pantalla la cantidad final que han acumulado con sus resultados parciales, además del tiempo de ejecución.

Como hemos visto, a cada thread se le asigna una función en la que trabajar y se le pasa una serie de argumentos: la posición en el array desde la que comienza a operar, su propio id y un puntero a la estructura de datos. Ahora, analizaremos las funciones sobre las que operan los threads en paralelo.

En cada una de las funciones, se realizan tres operaciones: calcular el resultado parcial del thread, si existe exceso y no está gestionado se opera y por último se actualiza el resultado global con el parcial. Tanto para comprobar si debe calcularse el exceso y acumular el resultado parcial, es imprescindible hacer uso del algún mecanismo de sincronización porque accedemos y modificamos variables compartidas.
Por ello, precisamos de un mutex para garantizar la exclusión mutua, **std::mutex mutex_threads**. Para bloquear y liberar el mutex, en ambos casos emplearemos ‘lock_guard’ en una sección crítica delimitada por scopes.

```
{
        std::lock_guard<std::mutex> lock(mutex_threads); //mutex lock
        if (datos->exceso_gestionado == false ){
                datos->exceso_gestionado = true;
                for (int i = 0; i < datos->tam_exceso; ++i){
                        thread_suma += datos->array_datos[(datos->size - datos->tam_exceso)+i];
```

```
                }
        }
}

...

{
        std::lock_guard<std::mutex> lock(mutex_threads);
        datos->resultado += thread_suma;
}
```

Cuando los threads acaban de computar en su sección de trabajo, comprueban si deben calcular el exceso. Es preciso asegurarnos de que solo un thread consulta dicha condición al tiempo, porque si no, estarían todos esperando a la liberación del mutex y lo calcularían de nuevo. Esta es la razón por la que establecemos una región crítica para dicha función.

Nuevamente, la estructura que acabamos de analizar se repite para cada función, según el tipo de operación.

Llegados a este punto, deberíamos tener un código funcional, capaz de operar un problema específico en paralelo. Para asegurarnos de que todos los resultados son correctos hemos ido añadiendo algunos puntos de depuración mediante compilación condicional.

```
#define DEBUG
...

#ifdef DEBUG
        printf ("El thread %d acaba primero y termina el exceso de trabajo.\n", id);
#endif

...
```

A medida que hemos ido progresando con el código, se han ido añadiendo distintos puntos de depuración con múltiples funciones y al final solo hemos dejado los que consideramos importantes. Inicialmente, al hacer multithreading se realizaba también la operación en secuencial para comprobar si el resultado obtenido de los threads era el deseado.

2.6. FEATURE_LOGGER

Una vez finalizada la parte base, continuamos con la versión FEATURE_LOGGER. Para hacer uso de esta característica del programa, hemos empleado compilación condicional. Su funcionalidad es recoger los resultados parciales de cada hilo, acumularlos él mismo y transferir el resultado al main, que comprobará si es válido. Para explicar cómo lo hemos diseñado, hablaremos sobre su creación y dependencias, comunicación thread-logger, funcionalidad logger, comunicación logger-main y contraste de resultados en main.

Si se seleciona la opción de logger, se tendrán que inicalizar una serie de campos en la estructura 't_data', además de los vistos previamente.

```
void inicializa_logger(t_data *datos, t_args *argumentos){

        datos->array_logger = (double*) malloc(argumentos->n_threads*sizeof(double));
        datos->thread_ready = false;
        datos->logger_ready = true;
        datos->main_ready = false;
        datos->resultado_logger = 0.00;
        datos->resultado_parcial = 0.00;
        datos->identificador_thread = - 1;
}
```

Lo esencial es que se dispone de un array de resultados parciales que el hilo logger irá completando y cuando finalice, los acumulará en 'resultado_logger'. Para este array se creará espacio dinámicamente de nuevo con malloc. El resto de campos, son relevantes para la comunicación entre los threads y el main.

Dada la inicialización, proseguimos con la creación del logger en el main antes de crear el resto de hilos.

```
#ifdef FEATURE_LOGGER
        datos.logger = std::thread(procesa_threads, &datos, &argumentos);
#endif
```

En cuanto a la comunicación thread-logger, explicaremos con detalle como se produce la sincronización y la transferencia de información.
Cuando los hilos terminan de computar están preparados para comunicar su resultado parcial al logger. Lo primero que se hace, es consultar si el logger se encuentra disponible, mediante un booleano. Si no lo está, el thread deberá ser dormido mediante una variable condicional. Si está disponible, el thread accederá a la sección crítica y procederá comunicando el resultado parcial e id al logger en la estructura 't_data'.
A continuación, el hilo despertará al logger para que recoja su resultado y liberará el mutex para que este lo tome. Seguidamente el thread acaba de computar.

```
#ifdef FEATURE_LOGGER

        std::unique_lock<std::mutex> ulk(mutex_threads_logger);
        if(datos->logger_ready == false){
                #ifdef DEBUG
                printf ("El thread %d espera a transferir su resultado al logger.\n", id);
                #endif
                cv_thread_logger.wait(ulk);
        }

        datos->logger_ready = false;
        datos->resultado_parcial = thread_resta;
        datos->identificador_thread = id;

        #ifdef DEBUG
        printf ("El thread %d ha comunicado al logger su resultado parcial e id.\n", id);
```

```
        #endif
        datos->thread_ready = true;

        #ifdef DEBUG
        printf ("El thread %d despertará al logger y liberará el mutex.\n", id);

        #endif

        cv_thread_logger.notify_all();
        ulk.unlock();

#endif
```

En base al código desarrollado, abordaremos una serie de aspectos. Cuando liberamos el mutex después de haber comunicado los resultados, debemos asegurarnos de que el logger siempre es el que lo bloquea después y no los siguientes hilos. Por esta razón, los siguientes threads que estén a la espera se tendrán que dormir y cuando el logger finalice, despertará a uno de ellos hasta que todos acaben. Esta es la solución que hemos encontrado ante el problema mencionado.

El hilo logger, también tendrá su propia función sobre la que trabajar, 'procesa_threads()'. Primero, rellena el array de resultados parciales hasta que todos los hilos terminen, acumula los resultados parciales según el tipo de operación y finalmente transfiere el cálculo al main.

```
int iterador = 0;

while(iterador < argumentos->n_threads){

std::unique_lock<std::mutex> ulk(mutex_threads_logger);
if(datos->thread_ready == false){
        cv_thread_logger.wait(ulk);

 }
datos->array_logger[datos->identificador_thread] = datos->resultado_parcial;


iterador ++;
datos->thread_ready = false;
datos->logger_ready = true;

cv_threads.notify_one(); //despertamos a un thread que se encuentre a la espera
ulk.unlock();

 }
...
```

Como vemos, los resultados parciales se almacenan ordenadamente y al terminar se despierta a uno de los hilos en espera de comunicar su resultado.
Finalmente, se acumula el resultado según la operación seleccionada y se comunica al main. En esta última parte, hacemos uso de otra variable condicional no global, en la que el main esperará a ser despertado para continuar.

```
{
        std::lock_guard<std::mutex> lock(mutex_logger_main);
        datos->main_ready = true;
}
        datos->cv_logger_main.notify_all();
```

Por último, cuando el thread sea despertado, determinará si el resultado recibido del logger es el mismo que el calculado por los hilos. También, imprimimos por pantalla si el resultado es correcto o incorrecto.

```
#ifdef FEATURE_LOGGER //si hemos usado logger

std::unique_lock<std::mutex> ulk(mutex_logger_main);

if(datos.main_ready == false){
        datos.cv_logger_main.wait(ulk); //dormir hasta que el logger lo despierte
}

ulk.unlock(); //liberamos mutex
datos.logger.join();

...

if (datos.resultado == datos.resultado_logger){
        resultado_correcto = true;
}

...


if (resultado_correcto){
        printf("El resultado de los hilos obtenido por logger es  correcto.\n”);

} else {
        printf("El resultado de los hilos obtenido por logger es incorrecto.\n");
 }

 std::cout<< "Valor recibido por logger: "<< datos.resultado_logger<<std::endl;
```

2.7. FEATURE_OPTIMIZE

En aquellos momentos donde podemos prescindir de mutex para modificar una variable compartida entre varios threads, podemos sustituir dicha variable para una atómica del mismo tipo para acelerar el proceso. Si analizamos nuestro código, existe una sección en particular donde si podemos hacerlo y es cuando los threads actualizan su resultado parcial con el global. De nuevo, para emplear esta característica usamos compilación condicional.

Para empezar, definimos la variable atómica en la estructura de datos ’t_data’, que después será incivilizada y accedida tanto por los threads como el main.

```
std::atomic<double> resultado_atm;
```

Cuando los threads tienen sus resultados parciales, pueden acumularlo en la variable atómica sin necesidad de emplear el mutex. Unicamente se revisa si la variable esta libre para modificarla y si lo está, se utiliza.

```
#ifdef FEATURE_OPTIMIZE
datos->resultado_atm.is_lock_free();
datos->resultado_atm = datos->resultado_atm + thread_suma;
```

Cuando los threads acaban de computar, el main debe discernir si se ha usado la variable atómica o no para comprar el resultado con el del logger y el mensaje a imprimir por pantalla.

```
#ifdef FEATURE_OPTIMIZE
if (datos.resultado_atm == datos.resultado_logger){
        resultado_correcto = true;
}

#else
if (datos.resultado == datos.resultado_logger){
         resultado_correcto = true;
}

#endif

...

#ifdef FEATURE_OPTIMIZE
        std::cout<<"Valor recibido por los threads: "<< datos.resultado_atm<<std::endl;
#else
        std::cout<<"Valor recibido por los threads: "<< datos.resultado<<std::endl;
#endif
```

3. Metodología y desarrollo de las pruebas realizadas.

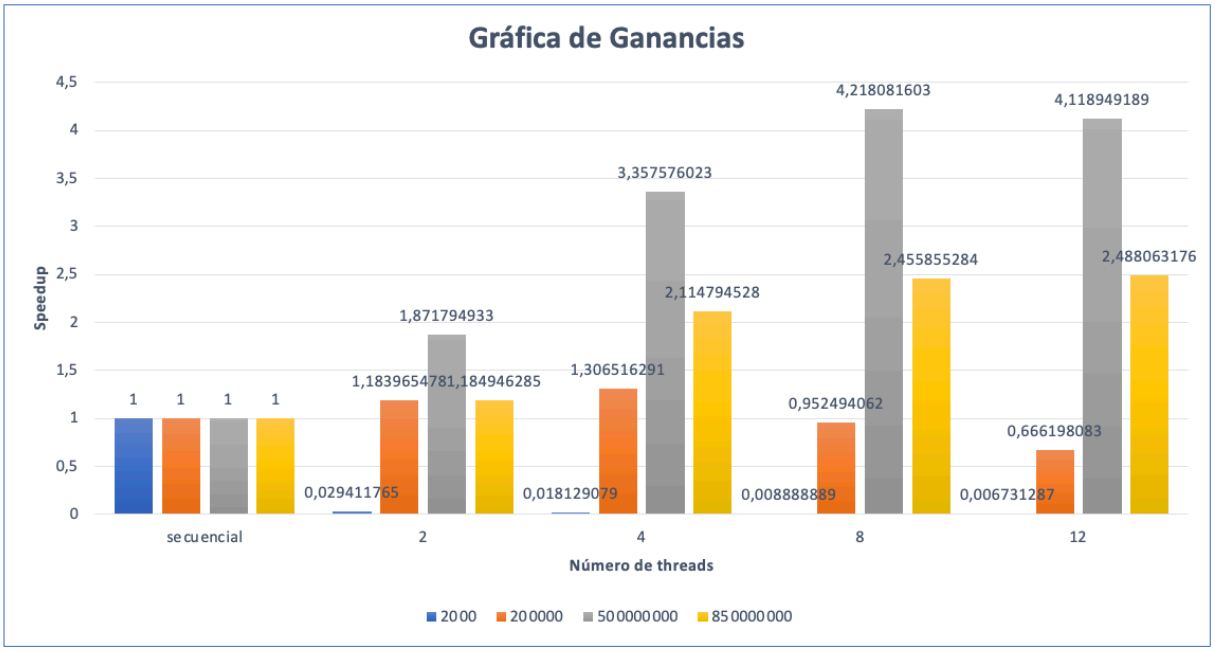
En este apartado expondremos el benchmarking realizado así como el análisis de los resultados obtenidos. Para empezar, hablaremos sobre las pruebas realizadas sobre la parte base de la práctica. Se contrastarán los resultados con 4 dimensiones distintas de array para determinar con precisión cuándo la ejecución paralela es mejor que secuencial, cuándo no supone ventaja y cuándo es significativamente peor. Además, nuestro interés será visualizar la tendencia de la ganancia y la eficiencia a medida que incrementamos progresivamente los tamaños. Después, se estudiarán los overheads generados en las operaciones de resta y xor frente a la suma. Por último, haremos distintas pruebas con la opción de optimización, para comprobar si de verdad implica mejora o no.

A la hora de anotar nuestros resultados, hemos decidido hacer uso de un script, ‘CLI_Script.sh’, para agilizar las pruebas y favorecer una leve pausa entre activaciones. Para construir gráficas fiables, realizaremos secuencias de 11 ejecuciones en cada caso, especificando el comando de ejecución. Se realiza al principio una activación añadida de calentamiento.

```
#! /bin/bash
for i in {0..10}
do
        sleep .05
        ./p1 20000 SUM single
done
```

Todos los resultados quedarán anotados en un fichero excel que se adjuntará con el código, 'resultadosCli.xlsx'. Para compilar nuestro programa, hemos utilizado: `'g++ -std=c++17 -pthread -o p1 p1.cpp -O3'.`

A continuación, comenzaremos analizando las pruebas realizadas.
Para poder determinar el rendimiento y escalabilidad del programa, haremos pruebas con distintos tamaños de matriz, en particular, 2000, 200000, 500000000 y 850000000 (máximo delimitado por Ram). De la misma forma, ejecutaremos con distintos números de threads: secuencial, 2, 4, 8 y 12. Seguidamente, abordaremos la gráfica de ganancias obtenidas. La ganancia se obtendrá dividiendo el tiempo secuencial entre el mejorado, en nuestro caso con el paralelo.

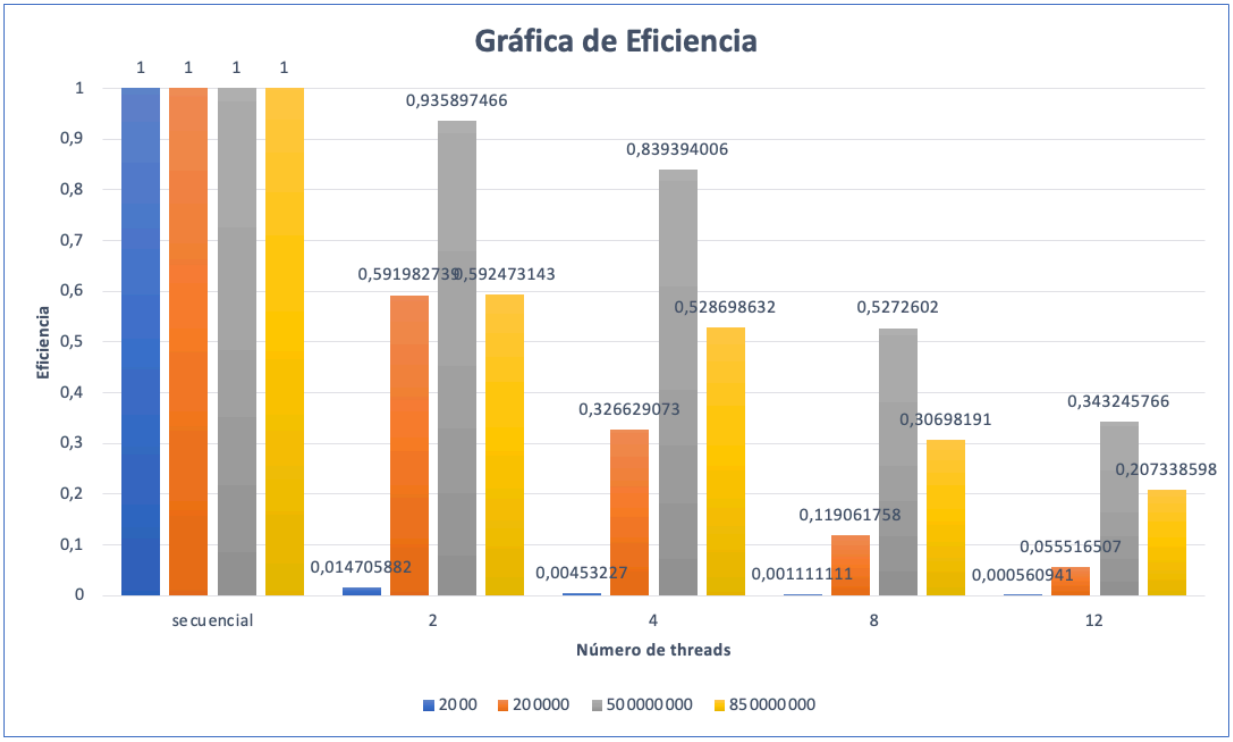


En base a los resultados generados, podemos extraer múltiples conclusiones. Para el primer tamaño de array y más pequeño (barra azul), es evidente que la paralelización no supone ningún tipo de mejora. La dimensión del array es muy limitada, solo la gestión que conlleva paralelizar supone una penalización en el tiempo considerable. Con el segundo tamaño de la muestra se comienza a apreciar alguna diferencia (barra naranja), la ejecución paralela con 2 y 4 threads implica un tiempo similar al secuencial pero no representa una ventaja llamativa. Esto se debe a que la reducción paralela empieza a suponer beneficio, aunque se vea limitado por el overhead que conlleva mantener varios hilos ejecutando a la vez.
Dando paso al tercer tamaño de la lista, se distingue una notable mejora (barra gris). El speedup comienza a crecer linealmente hasta los 4 threads y luego alcanza un máximo de 4,21 donde se estabiliza y comienza a descender. Nuevamente, la gestión paralela y

los desequilibrios de carga impiden un rendimiento progresivo hasta la ejecución con 4 hilos. Por último, el cuarto y último tamaño a comentar establece ganancias menores comparadas con el anterior caso (barra amarilla). A estas alturas, la penalización de tiempo que conlleva la gestión en paralelo es excesiva, y por lo tanto la ganancia será cada vez menor si continuamos aumentando las dimensiones del array.
Es preciso mencionar también que al disponer de una máquina con 8 cores, cuando añadimos más hilos de los correspondientes, nuestro speedup siempre se estabilizará y descenderá porque los componentes hardware (ALUs) comenzarán a ser compartidos.

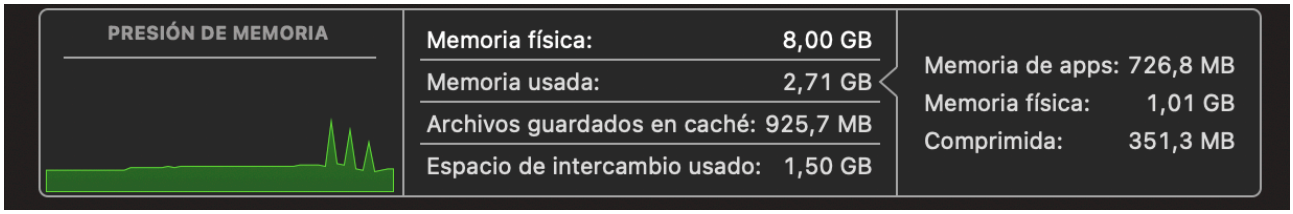
La conclusión que podemos extraer de esta gráfica es que paralelizar no siempre supone una gran mejora. Cuando nuestro problema a computar es demasiado pequeño o muy grande, no compensa este modo de ejecución. Quizás obtengamos una mejora aceptable pero el esfuerzo que conlleva no es abordable. Sin embargo, hay momentos en los que si vale la pena, como el caso anterior, donde la reducción paralela favorece el tiempo de ejecución y no se ve sometida en gran medida por la gestión de los hilos.

Ahora, analizaremos la gráfica de eficiencia en base a los resultados anteriores. La eficiencia surge de dividir el speed up real entre el teórico. El speedup teórico es el número de threads que computan en paralelo.



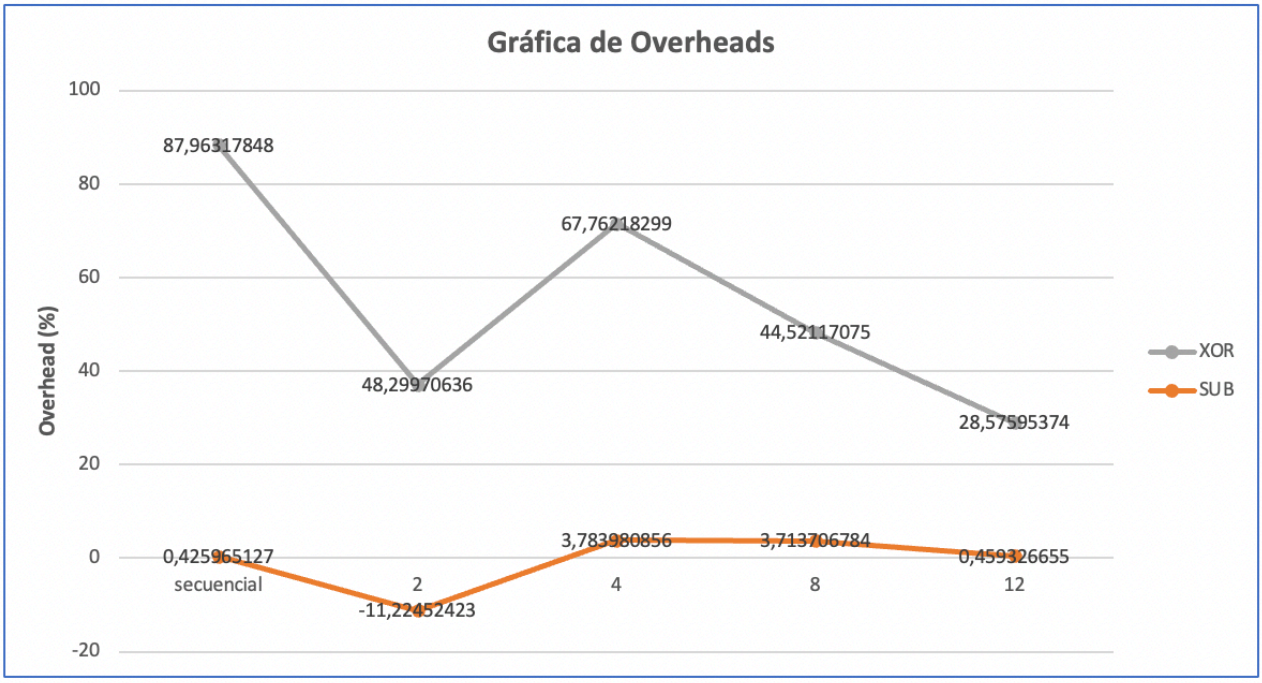
Como ya hemos visto antes, el tercer tamaño de array, con el bloque grisáceo, representa el mejor rendimiento respecto del resto de dimensiones. También, conforma la mayor eficiencia con cualquier número de threads ejecutando. En los instantes donde se paraleliza con 2 y 4 threads, el speedup obtenido es casi igual al teórico, la eficiencia es muy cercana a 1. Esto implica que la mejor paralelización se consigue con menos hilos, para reducir tanto el overhead como los desequilibrios producidos y optimizar la ejecución del código. La tendencia global te la eficiencia es una curva en descenso, indicando que cuantos más hilos, menor será nuestro beneficio.

El segundo benchmarking realizado, hace referencia a los overheads de las operaciones resta y xor respecto de la suma. En este caso, para comparar los resultados, escogeremos el mayor tamaño posible según las dimensiones de nuestra Ram.
Para averiguar el espacio libre que disponemos en nuestra caché, después de investigar mucho, no hemos dado con un equivalente del comando en Linux ‘free -m’ para Mac. Por lo tanto, hacemos uso de una herramienta ya instalada en nuestro ordenador llamada “Monitor de Actividad”. En el apartado de memoria, visualizamos el siguiente contenido.



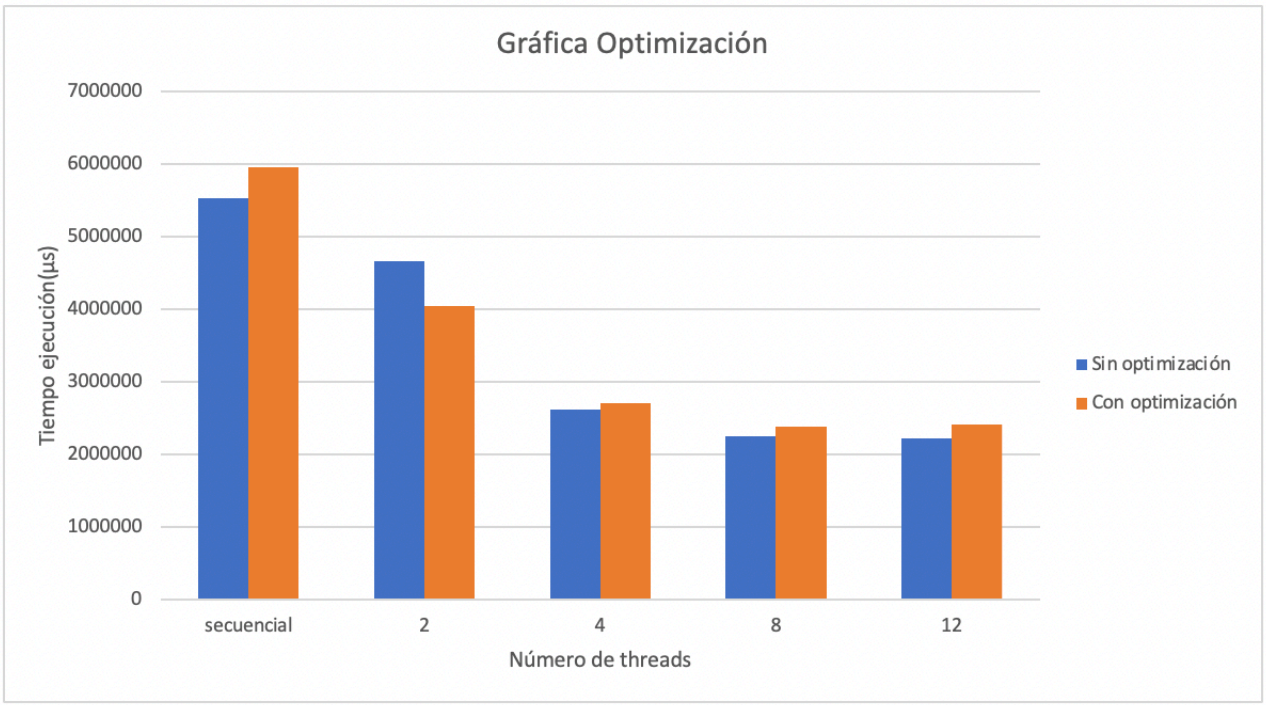
Lo que ya conocíamos eran los 8gb de espacio, pero no sabíamos cuánto de esa memoria estaba ocupada. Como vemos, según los “Archivos guardados en caché”, hay 1gb aproximadamente. Para no andar completamente justos, consideraremos que hay 6,5gb libres de memoria. Entonces, traduciremos todo ese espacio en elementos que podemos usar para nuestro array. Multiplicaremos 6,5*1000 para convertir a Kb y después en bytes, es decir, 6500*2^20 bytes. Puesto que estamos trabajando con doubles, dividiremos entre 8 porque son los bytes que ocupa cada elemento. Entonces, obtenemos un tamaño aproximado de 850000000, el cual hemos utilizado en la prueba anterior y cuyo comportamiento ya conocemos.

Seguidamente, continuaremos con los overheads de los algoritmos. Para cada operación, se han realizado 5 pruebas de 10 ejecuciones, con los números de threads antes utilizados. Una vez disponiendo de los tiempos medios, el overhead de la resta y xor surge de la fórmula ((Ta/Tb)-1,0)*100 = X%, siendo Tb el tiempo medio obtenido con suma. Calculados los resultados, elaboramos una gráfica que analizaremos detenidamente.



A primeras, es evidente que la operación xor implica un overhead mucho mayor que la resta respecto de la suma. Después de buscar información, podemos determinar que no necesariamente la suma debe ser más rápida que xor, simplemente según nuestra arquitectura y las dimensiones del problema, obtiene tiempos de ejecución peores. Por ejemplo, en secuencial es casi el doble del tiempo que conlleva xor frente a la suma. En cualquiera de los casos, xor con este tamaño es más lento. En cuanto a la resta, los tiempos aparecen muy similares y es entendible. Al final, la resta es una suma con elementos negativos. En todos los casos, la suma casi no supone overhead.
La gráfica también nos proporciona más información y salta a la vista. Como podemos observar, con dos threads computando en paralelo, el overhead en las dos operaciones disminuye mucho, tanto que en el caso de la resta es aún más rápida que la suma.
La conclusión que podemos formar es que con este número de hilos, con este tamaño de array y con todos los tipos de operaciones se obtienen resultados mejores.

Finalmente, la tercera y última prueba que detallaremos será el uso de la optimización. Tal y como se ha explicado en el diseño del código, la optimización que hemos considerado ha sido convertir aquellas variables compartidas que precisaban de un mutex para ser modificadas, en atómicas y así prescindir de dicho mutex precisamente. En nuestro caso, esta condición solo se favorece cuando los threads actualizan la variable global con su resultado parcial. Por esta razón, suponemos que no observaremos gran ventaja al emplear optimización.



Con un tamaño de 850000000, a simple vista no hay mucha diferencia. Podemos destacar además que sin optimización se consiguen tiempos mejores salvo con 2 threads, donde la optimización presenta ventaja. Con el resto de threads que hemos probado, la tendencia es la misma, la optimización empeora levemente el resultado. Efectivamente, podemos concluir que en nuestro caso la optimización no supone gran ventaja. Cabe destacar, que si accediésemos múltiples veces para modificar una variable compartida entre varios threads, podría acabar notándose un cambio al prescindir de mutex.

4. Valoración

Desde mi punto de vista, sin esta práctica el resto no tendrían sentido. Es muy extensa y completa pero ayuda a entender muy bien cómo funciona la paralelización y la sincronización de los threads internamente.
Haciendo esta actividad, independientemente de los conocimientos nuevos adquiridos, he apreciado la importancia de hacer pseudo-códigos mejores para conseguir que el programa realice exactamente lo que necesite y considero que es algo importante y valioso de cara al futuro. También me parece relevante todo el proceso que conlleva el diseño pues primero nos hemos tenido que asegurar del funcionamiento para poder continuar y pasar a la siguiente parte. La depuración también fue esencial de principio a fin y pienso que me siento más ágil también en ese aspecto. Por último, el momento de experimentación con benchmarking resultó llamativo pues hemos adoptado soltura con otras herramientas para analizar resultados mediante gráficas detalladas.
Me habría gustado tener más tiempo para completar las partes opcionales pero lo que he llegado a terminar, lo he disfrutado mucho.