Desarrollo del Laboratorio 7 - 20213852

1. **(1.0 punto) Incluir la medición de tiempo desde C considerando únicamente el cálculo de la suma para ambos programas. Imprimir el tiempo de ejecución al ejecutar los programas. Utilice las unidades que mejor se ajusten al tiempo de ejecución de su PC. No usar más de dos decimales.**

home > labtel > Descargas > C Programa1_1.c > main()

```c
#include <stdio.h>
#include <time.h>
            (int)1000000
#define N 1000000

int main() {
  struct timespec ti,tf;
  double elapsed;

  int sum = 0;
  int array[N];

  // Initialize the array
  for (int i = 0; i < N; i++) {
    array[i] = i;
  }
  clock_gettime(CLOCK_REALTIME,&ti);
  // Calculate the sum of the array
  for (int i = 0; i < N; i++) {
    sum += array[i];
  }
  clock_gettime(CLOCK_REALTIME,&tf);
  elapsed=1e9*(tf.tv_sec-ti.tv_sec)+tf.tv_nsec-ti.tv_nsec;

  // Print the time
  printf("El tiempo que toma el programa es %.2lfns\n",elapsed);

  return 0;
```

home > labtel > Descargas > C Programa1_2.c > main()

```c
#include <stdio.h>
#include <time.h>

#define N 1000000

int main() {
  struct timespec ti,tf;
  double elapsed;
  int sum = 0;

  // Calculate the sum of the array
  clock_gettime(CLOCK_REALTIME,&ti);
  for (int i = 0; i < N; i++) {
    sum += i;
  }
  clock_gettime(CLOCK_REALTIME,&tf);
  elapsed=1e9*(tf.tv_sec-ti.tv_sec)+tf.tv_nsec-ti.tv_nsec;

  // Print the time
  printf("El tiempo que toma el programa es %.2lfns\n",elapsed);

  return 0;
}
```

```
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1758475.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1738741.00ns
```

2. **(1.0 punto) Realizar 15 ejecuciones desde el terminal para ambos archivos ejecutables. Realizar una captura de pantalla de su ejecución. Los tiempos en su imagen deben ser visibles.**

```
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1758475.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1686817.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1759079.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1720535.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1715992.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1665421.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1733029.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1760865.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1759911.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1853140.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1769558.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1754576.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1725293.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1725687.00ns
labtel@localhost:~/Descargas$ gcc Programa1_1.c -o Programa1_1 && ./Programa1_1
El tiempo que toma el programa es 1735226.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1738741.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 2043348.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1629501.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 &&
El tiempo que toma el programa es 1845142.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1881013.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1671980.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1756055.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1801600.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1714714.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1622546.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1722612.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1637802.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1812619.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 1832385.00ns
labtel@localhost:~/Descargas$ gcc Programa1_2.c -o Programa1_2 && ./Programa1_2
El tiempo que toma el programa es 2044948.00ns
```

3. **(1.0 punto) Adjuntar una tabla con todas las mediciones.**

Medición de tiempo (Programa1_1.c) en ns	Medición de tiempo (Programa1_2.c) en ns
1758475.00	1738741.00
1686817.00	2043348.00
1759079.00	1629501.00
1720535.00	1845142.00
1715992.00	1881013.00
1665421.00	1671980.00
1733029.00	1756055.00
1760865.00	1801600.00
1759911.00	1714714.00
1853140.00	1622546.00
1769558.00	1722612.00
1754576.00	1637802.00
1725293.00	1812619.00
1725687.00	1832385.00
1735226.00	2044948.00
1741573.6 (media)	1783667.07 (media)

4. **(1.0 punto) Explicar cómo se relaciona el Programa1_1.c con los conceptos de localidad temporal y localidad espacial.**

Podemos dividir la explicación y mencionar cómo cada variable encontrada en el programa (fuera de las usadas para medir el tiempo e imprimir) se relaciona con los conceptos de localidad temporal y localidad espacial.

sum: Esta variable se declara como **int** al comenzar el programa 1 y se le asigna el valor ‘0’. ***sum*** vuelve a ser encontrada dentro del segundo bucle **for**, en donde lee y se sobreescribe su valor por cada iteración que se realiza. Esto implica que por cada iteración se vuelva a reutilizar la misma dirección de memoria (***sum***) y se logre afianzar una fuerte localidad temporal.

array[N]: Este arreglo tiene un conjunto de **N** términos que son de tipo **int**. Se declara el tipo de arreglo y la cantidad de memoria necesaria para todos los datos al iniciar el programa. Dentro del primer bucle **for** notamos que acabamos accediendo a todas las direcciones de memoria de los términos del arreglo **array** e ingresando valores, de forma ascendente y una por cada iteración. Comenzamos accediendo a **array[0]**, para seguir con **array[1], array[2]** y así sucesivamente. Después de acceder a **array[0]**, accedemos a términos que conforman parte o se encuentran muy cerca del espacio de este, por lo que se puede denotar una fuerte localidad espacial. Dentro del segundo bucle **for** accedemos nuevamente a cada una de las direcciones de memoria de los términos de este arreglo, una por iteración y de forma ascendente, pero esta vez solo para realizar lectura. De la misma forma, comenzamos accediendo a **array[0]** para después acceder términos que

conforman parte de su espacio, como **array[1], array[2]**, etc. Por ello, aquí también se revela una fuerte localidad espacial.

i(contador): Esta variable se declara e inicializa con valor '0' dentro de cada bucle **for**. Sin embargo, su valor solo aumenta de 1 en 1 por lo que forma parte de un registro manipulado por un adder, y no podría acceder a algún tipo de localidad empleada en memorias.

5. **(1.0 punto) Explicar cómo se relaciona el Programa1_2.c con los conceptos de localidad temporal y localidad espacial.**

De la misma forma que el inciso anterior, podemos dividir la explicación de la relación partiendo por cada una de las variables que conforma el programa.

sum: Esta variable se declara como **int** al inicio del programa 2 y se le asigna el valor '0'. Después la volvemos a apreciar dentro del primer y único bucle **for** presente en el programa, en donde se lee y se sobreescribe continuamente su valor por cada iteración que se realiza. Esto implica que se vuelva a reutilizar la misma dirección de memoria (***sum***) y se logre afianzar una fuerte localidad temporal.

i(contador): Corresponde al mismo caso del contador del programa 1. Después de declarar e inicializar su valor '0' dentro del bucle **for**, solo aumenta su valor de 1 en 1 condicionado a funcionar como registro y no beneficiarse de algún tipo de localidad empleada en memorias.

6. **(1.0 punto) Utilizar el comando getconf -a | grep CACHE en el terminal y verificar el tamaño de bloque de su computador. Recordar que el tamaño de bloque en el nivel 1 se ve en la línea LEVEL1_DCACHE_LINESIZE (en bytes). Adjuntar captura de pantalla.**

```
labtel@localhost:~/Descargas$ getconf -a|grep CACHE
LEVEL1_ICACHE_SIZE                 32768
LEVEL1_ICACHE_ASSOC
LEVEL1_ICACHE_LINESIZE             64
LEVEL1_DCACHE_SIZE                 32768
LEVEL1_DCACHE_ASSOC                8
LEVEL1_DCACHE_LINESIZE             64
LEVEL2_CACHE_SIZE                  262144
LEVEL2_CACHE_ASSOC                 4
LEVEL2_CACHE_LINESIZE              64
LEVEL3_CACHE_SIZE                  16777216
LEVEL3_CACHE_ASSOC                 16
LEVEL3_CACHE_LINESIZE              64
LEVEL4_CACHE_SIZE                  0
LEVEL4_CACHE_ASSOC
LEVEL4_CACHE_LINESIZE
```

El tamaño de bloque del computador es de 64 bytes.

7. **(2.0 punto) Conociendo los datos previos, ¿es posible saber en qué momento se produce hit o miss? De ser posible, comentar el ejemplo, de no serlo, explicar porqué.**

Efectivamente. Según los programas mostrados es posible saber en qué momento se produce algún hit o miss o al menos nos podemos dar una idea de cada cuánto sucede. Sea el caso en que las memorias Caché L1, L2, L3 y la memoria RAM se encuentren vacías o llenas (sin contener alguna dirección de memoria dentro relacionada a alguno de los programas) se puede estimar que:

Cantidad de bytes disponible: L1=32768

L2=262144

L3=16777216

Cantidad de conjuntos: L1=4096

L2=32768

L3=2097152

Cantidad de bloques: L1=512

L2=4096

L3=262144

Cantidad de valores int almacenables por bloque: 64 bytes / 4 bytes = 16

Para el programa 1:

Cantidad de bytes total a usar = (sizeofint)*(1 (sum) + N(1000000 de array[N])) = 4 * 10000001 = 4000004 bytes

1. Primero se declaran ***sum*** y ***array[N]***, los cuales tendrán asignados direcciones de memoria dentro de la RAM, junto a su valor, si este es especificado. ***sum*** conforma su propio bloque, ya que ***array[N]*** recién es declarado después. Continuemos analizando el primer bucle **for**.

2. Notamos que la memoria caché L3 puede contener absolutamente todos los elementos del array necesarios debido a que su tamaño es superior al total empleado por array[N]. Esto quiere decir que al momento de colocar los bloques de la memoria en esta memoria caché, se producirán miss hasta que finalmente todos los elementos del array se encuentren aquí alojados. Posteriormente, el hit ratio será de 100%.

3. La memoria caché L2 no puede contener a todos los elementos necesarios, por lo que mediante el algoritmo empleado (supongamos en este caso FIFO, puesto que fue el único trabajado en clase) se sobreescribirán los bloques que estén asociados a elementos que hayan sido empleados primero, como podrían ser los primeros elementos del array. Así como para la caché L3, se producirán tantos miss por bloque que sean necesarios hasta que la memoria caché L2 se llene. Cuando esto suceda, los bloques que hayan sido inicialmente trasladados aquí como ***array[0]-array[15]***, ***array[16]-array[31]***, etc, se sobreescribirán con los nuevos bloques correspondientes al valor de ***i*** actual en donde se llene la memoria caché L2. Al ser los bloques antiguos ya inservibles, no afectan al hit ratio que esté presentando la memoria caché L2.

4. Con la memoria caché L1 se repite lo mismo pero de una forma aún más pronunciada. Ya que el tamaño de esta memoria es aún más pequeño, se llenará más rápido. Hasta tal entonces, se producirá un miss por cada nuevo elemento del array que forme parte de un nuevo bloque que no haya sido asignado previamente en esta memoria caché. Podemos realizar el siguiente análisis y replicarlo para cada una de las memorias caché (ya que el sobreescribir bloques no afecta al hit

ratio, al menos para este bucle for): Al momento de buscar la dirección de memoria ***array[0]*** en la caché L1 no se encuentra, por lo que se busca en la caché L2 y al no estar tampoco aquí en la caché L3. Al no estar en ninguna de estas pero sí en la memoria, se copia el bloque junto a todos sus elementos a todas las memorias caché; es decir, desde ***array[0]*** hasta ***array[15]***. Por ello, al posteriormente buscar la dirección de memoria ***array[1]*** se producirá un hit en la caché L1. Esto se repetirá hasta llegar a ***array[15]***, para posteriormente producir un nuevo miss y repetir el mismo bucle por siempre. Como mencioné antes, esto se repite para la caché L2 y L3. Podemos estimar que existe entonces alrededor de un 15/16 * 100% = 93.75% hit ratio.

5. Para el segundo bucle **for** notamos que ahora contamos con la memoria caché L3 conteniendo todos los bloques necesarios para poder acceder a todos los elementos del array, pero sin la dirección y valor de ***sum***. Por ello, se produce un miss para después por cada iteración que se realice producir un hit al referir a la variable ***sum***.

6. Como esta variable cuenta con su propio bloque, ahora contamos con 1 bloque para ***sum*** y 511 para los elementos del array dentro de la caché L1. Sin embargo, ello no afectará su hit ratio, ya que este no depende de la cantidad de bloques que se tenga debido a lo mencionado en (3).

7. Cuando se comience a buscar los bloques que contengan a los primeros elementos del array en la caché L1 no se encontrarán, puesto que la última vez que se accedió aquí se llegaron a almacenar los últimos bloques del array como ***array[999984]-array[999999]***. Por ello, se tendrá que buscar estos primeros elementos ahora en la caché L2. Sin embargo, aquí ocurre lo mismo, por lo que finalmente se encontrarán estos elementos en la caché L3, en donde si se consiguieron alojar todos los elementos de array. Por ello, se repite el mismo patrón que en el primer bucle **for**: 1 miss por cada nuevo elemento que no se encuentra y 15 consiguientes hit por cada elemento que se encuentre dentro del bloque del mismo elemento que produjo el miss. En total, para la caché L1: 1 miss para ***sum*** + 511*1 miss para cada bloque nuevo del array = 512 miss. Por otro lado: 511*16-1 hit para ***sum*** (hasta que su bloque sea reemplazado) + 511*15 hit para los elementos cercanos al elemento que produjo el miss. En total, el hit ratio aumenta a aprox 96.9%.

Para el programa 2: Cantidad total de bytes a usar: 4 (solo para int ***sum***).

Después de haber declarado y asignado el valor ‘0’ a ***sum***, se procede a trasladar su bloque de la RAM a las 3 memorias caché, debido a que no se encontraba en ninguna. Posteriormente, se reutilizará este mismo bloque para los siguientes iteraciones necesarias. Habrá un único miss en las 3 memorias caché debido a la primera búsqueda del valor ***sum.*** Todos los intentos restantes corresponderán a un hit debido a que solo se mantendrá ese único bloque durante todo el programa sin tener que trabajar con más bloques y que esto posibilite que el bloque de ***sum*** se reemplace por otro debido al algoritmo **FIFO**, por lo que habrá un 999999/1000000 * 100% = 99.9999% hit ratio.

8. **(4.0 puntos) Basándose en todas las respuestas de los incisos anteriores, ¿qué implementación genera mejores tiempos de ejecución? Justificar su respuesta.**

La implementación que genera mejores tiempos de ejecución es la del primer programa. Podemos justificar esto debido a la presencia de una fuerte localidad espacial (por la presencia del array) aparte de la localidad temporal ya existente (***sum***). El segundo programa solo gozaba de una fuerte localidad temporal (***sum***).

9. **(3.0 puntos) ¿Cómo influye el tipo de datos del arreglo en este ejercicio? Esperaría resultados similares para un tipo char, short, long? Comentar acerca de su respuesta.**

Un diferente tipo de datos para el arreglo modifica la cantidad de elementos que se pueden asignar por bloque dentro de la caché, debido a que cada bloque contiene una cantidad de bytes fija. Al haber declarado a ***array[N]*** como char (considerando que no existiera overflow), la cantidad de elementos posibles para encajar en un bloque serían de 64 bytes / 1 byte = 64, por lo que las memorias caché se llenarían de una forma más lenta, y se conseguiría un mayor hit ratio. Esto debido a que después de haber producido un miss al intentar acceder a ***array[0]***, tendríamos en la caché L1 el bloque respectivo hasta ***array[65]***. De esta forma, se evaluaría un 63/64 * 100% = 98.44% de hit ratio, lo cual mejoraría el tiempo de ejecución del programa 1.
Al haber declarado el array como short, hubiera ocurrido también una optimización en el tiempo de ejecución debido a que cada bloque podría haber albergado 64 bytes / 2 bytes = 32 elementos. Esto también ignorando el overflow existente al tener solo 2 bytes para relacionar números gigantes.
Por el contrario, al usar long se hubiera notado un mayor tiempo de ejecución en el programa, debido a que solo se podría haber alojado 64 bytes / 8 bytes = 8 elementos por bloque. Esto hubiera provocado que ocurra una menor cantidad de hit después de ocurrir un miss al acceder a un valor que no estuviese dentro de la caché. Podríamos aproximar un hit ratio de: ⅞ * 100% = 87.5%.